ZX Spectrum +3

Contents

Introduction

Sinclair ZX Spectrum +3
128K Integrated Home Computer/Disk System

Following on from the outstanding success of the established ZX range of computers, the original Spectrum, the Spectrum +, the Spectrum 128 and the new generation Spectrum +2, we now proudly present the ZX Spectrum +3, a machine that combines the very best features of the previous Sinclair models, with the added convenience of a fast access floppy disk drive.

The whole is a truly complete computer/disk system which allies established Sinclair technology with AMSTRAD's expertise in integration and engineering reliability, and flair for producing a 'no nonsense' all-in-one package.

Software compatibility

The +3 may be used with software written for the earlier models in the ZX Spectrum range. This means that a vast quantity of software already exists for the +3. There are literally thousands of titles available covering every conceivable application: games, utilities, music, scientific, educational and many many more.

The BASIC

The +3 uses a computer language called BASIC (Beginners' All-purpose Symbolic Instruction Code). BASIC is by far the commonest language for home computers, and +3 BASIC has been designed to be particularly easy to learn and use.

How to read this book

In order to get the best out of your +3, it is vital that you read all the relevant information provided in this manual. If you skip various sections, it is likely that you will come to a grinding halt later on!

Therefore, you should adopt the following reading programme:

Chapter 1 - This chapter shows you how to connect up your +3 system. Note especially the safety warnings regarding the wiring-up of the mains plug.

Chapter 2 - This chapter describes the switching on of the +3 and shows you how to tune-in your TV to display the computer's signal. You are then shown how to select an option from the 'opening menu' -and if you don't know how to do that, you'll not be able to use the +3 at all! If, however, you do know how to tune-in your TV and select menu options (perhaps by having previously used a Spectrum 128 or a +2), then you may skip this chapter.

Chapter 3 - This chapter shows you how to load commercially available disk software. If you never intend to use such software, then you may skip this chapter.

Chapter 4 - This chapter shows you how to load commercially available pre-recorded tape software. If you never intend to use such software, then you may skip this chapter.

Chapter 5 - This chapter covers the use of the ***+3***'s built-in disk drive (known as drive A:). You may skip this chapter only if you never intend use the disk drive during BASIC programming (perhaps having purchased the ***+3*** solely to load and run commercially available software (eg. games)). Note that if you have connected an additional disk drive (B:) to the ***+3***, then throughout this manual you should take any general references to 'the disk drive' as meaning both drives (A: and B:).

Chapter 6 - This chapter introduces you to ***+3*** BASIC. In particular, it describes the editor and certain aspects of BASIC programming that differ from those of other computers. Therefore, even if you are an experienced BASIC programmer on another computer, you should still read chapter 6. Note that you'll require a blank CF-2 floppy disk as you work through this chapter. If, however, you never intend to program in BASIC and have purchased the ***+3*** solely to load and run commercially available software (eg. games), then you may skip this chapter.

Chapter 7 - This is the one chapter that you may freely skip. It describes the 48 BASIC mode (in which the ***+3*** operates exactly like the 'old-style' Spectrum - even in the editing and programming aspects). This mode is not recommended for anything other than a history lesson for the curious, or for loading old (Spectrum 48 only) tape software. You should certainly not use this mode for BASIC programming; indeed you cannot access many of the advanced features of the ***+3*** (including disk drive, extra memory, **RS232/MIDI/AUX** interfaces or RAMdisk) from 48 BASIC. Notwithstanding the above, we have provided the relevant information in this chapter for your reference.

Chapter 8 - This chapter forms the very heart of the manual. It is a complete guide to BASIC programming on the ***+3***. If you have programmed in BASIC before, then you may wish to use this chapter merely as a reference guide, searching the main index to find the information you need from one of the subsections. If, on the other hand, you are new to BASIC, you may wish to work through the chapter, one subsection at a time, developing your programming skills as you go. Once you are able to type in and run a program, and have grasped a few of the fundamentals of BASIC, then you may feel confident about skipping ahead to later subsections. If, however, you never intend to program in BASIC and have purchased the ***+3*** solely to load and run commercially available software (eg. games), then you may skip this chapter.

Chapter 9 - This chapter shows you how to use the ***+3*** as a calculator only. You may skip this chapter if you wish.

Chapter 10 - This chapter illustrates how add-ons (peripherals) are connected to the ***+3***. Peripherals include such devices as a cassette unit, a printer, an additional disk drive, a joystick, etc. So if you're thinking of linking up any device at all to the ***+3***, check this chapter to make sure that you've got the right connections. If, on the other hand, you intend to use just the standard ***+3*** set up (ie. computer and TV only), then you may skip this chapter.

AMSTRAD

All correspondence relating to the product or to this manual should be addressed to:

Sinclair Computers Division
AMSTRAD Plc
Brentwood House
169 Kings Road
BRENTWOOD
Essex CM14 4EF

First Published 1987

Written by Ivor Spital
Contributions by Cliff Lawson and Rupert Goodwins
Produced by Des Rackliff

Extracts from the book 'ZX Spectrum BASIC programming' written by Steven Vickers and Robin Bradbeer

Published by AMSTRAD

+3DOS written by Locomotive Software Ltd.

Acknowledgements to Centronics and Epson Corps.

Precautions!

You must read this....

(Don't worry if you are a little baffled by some of the technical jargon in this section: the importance of these warnings will become clearer as you work through this manual.)

1. Always connect the mains lead of the power supply unit (PSU) to a 3-pin plug following the instructions given in chapter 1.
2. Do not attempt to connect the PSU to any mains supply other than 220-240V AC 50Hz.
3. After you have finished using the ***+3*** always disconnect the PSU from the mains supply socket.
4. There are no user serviceable parts inside the equipment - DO NOT ATTEMPT TO GAIN ACCESS INSIDE THE PSU - THERE ARE HIGH VOLTAGES INSIDE. Refer all servicing to qualified service personnel.
5. Do not block or cover the ventilation slots in the equipment.
6. Do not use or store the equipment in excessively hot, cold, damp, or dusty areas.
7. Never plug in (or unplug) any device from any of the rear sockets while the ***+3*** is switched on - doing so will probably damage both the ***+3*** and the device.
8. Never switch the ***+3*** on or off while a disk is inserted in the disk drive. Doing so may corrupt your disk, losing valuable programs or data.
9. After you have switched off your TV (or VDU monitor), do not immediately disconnect the ***+3*** - wait a few seconds or so.
10. Do not switch off the ***+3*** (or switch on or off any peripheral devices connected to the ***+3***) while there is a program or data in the memory that you wish to keep - doing so may make the ***+3*** 'crash', losing the program or data.
11. Always keep the disk drive and disks away from magnetic fields. For maximum data reliability, do not position the disk drive close to your TV or monitor, or close to any source of electrical interference.
12. If you have connected an additional disk drive to the ***+3***, keep the ribbon cable (to the additional drive) away from mains leads.
13. Whenever possible, make back-up (duplicate) copies of disks which contain valuable programs. Otherwise, should you accidentally lose or corrupt the disk, replacing it may prove very expensive.
14. Never touch the floppy disk surface itself, inside its protective casing.
15. Do not eject a disk while it is being read from or written to.
16. Always remember that formatting a disk will erase its previous contents.

Chapter 1
Open the box

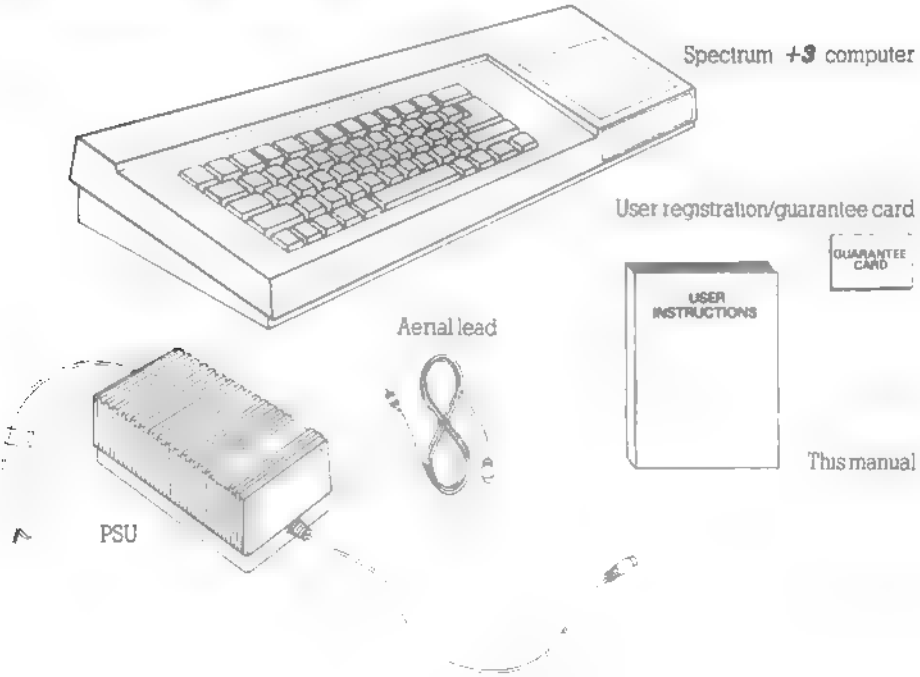
Subjects covered...

Unpacking
Fitting a mains plug
Setting up

Unpacking

Inside the carton, you'll find the following...

The Spectrum +3 computer
The power supply unit (PSU)
The aerial lead
This manual (together with your user registration/guarantee card)



Fitting a mains plug

The power supply unit for the Spectrum *+3* operates from a 220-240 Volt AC 50Hz mains supply.

Fit a proper mains plug to the mains lead of the power supply unit. If a 13 Amp (BS1363) plug is used, a 3 Amp fuse must be fitted. The 13 Amp fuse supplied in a new plug must NOT be used. If any other type of plug is used, a 5 Amp fuse must be fitted either in the plug or adaptor or at the distribution board.

IMPORTANT - The wires in this mains lead are coloured in accordance with the following code...

Blue : Neutral

Brown : Live

As the colours of the wires in the mains lead of this apparatus may not correspond with the coloured markings identifying the terminals in your plug, proceed as follows...

The wire which is coloured BLUE must be connected to the terminal which is marked with the letter **N** or coloured black.

The wire which is coloured BROWN must be connected to the terminal which is marked with the letter **L** or coloured red.

Disconnect the mains plug from the supply socket when not in use.

Do not attempt to remove any screws, nor open the casing of the power supply unit. Always obey the warning on the rating label of the power supply unit...

WARNING: LIVE PARTS INSIDE - DO NOT REMOVE ANY SCREWS

Setting up

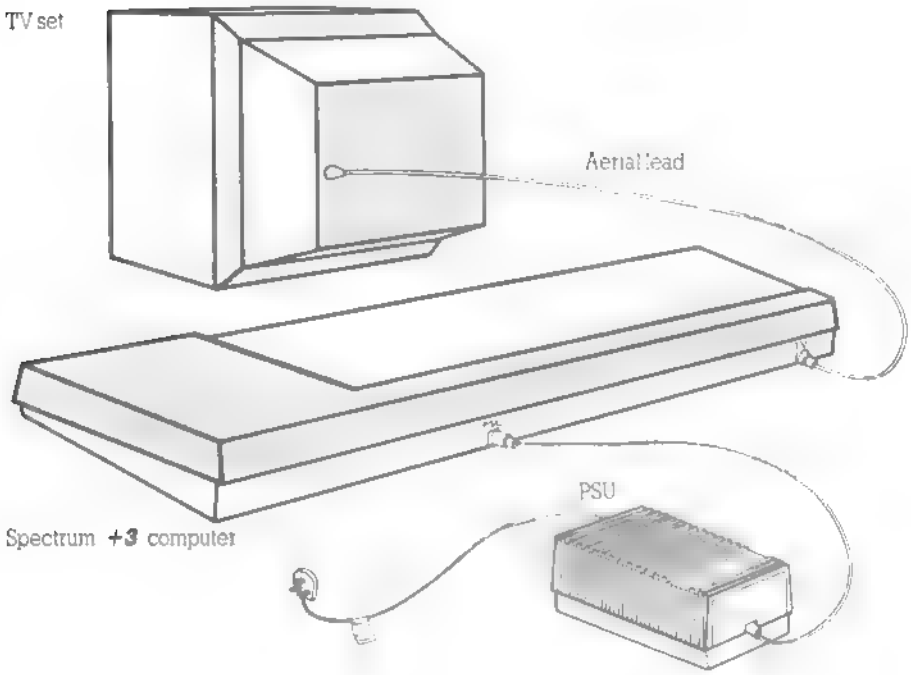
We will now set up the standard *+3* system. All you need (other than the items you unpacked) is a standard TV set (UHF). You can use a colour or black-and-white TV, but of course, with the latter you will not be able to enjoy the full colour capabilities of your *+3*.

Note that if you wish to attach one or more add-ons, or ***peripherals***, (eg. printer, joystick, cassette deck, second disk drive, monitor, audio amplifier, MIDI device, modem or other serial/expansion device) to your *+3* system, you should turn to chapter 10 (Peripherals for your *+3*).

Place the *+3* computer on a suitable flat surface, ready to be connected to your TV. Next, remove any plug which is already connected to the aerial socket at the back of the TV. Using the aerial lead provided with your *+3*, insert the larger plug into the TV's aerial socket, and insert the smaller plug into the socket marked **TV** at the back of the *+3*.

Finally, insert the 6-pin DIN plug coming from the power supply unit into the socket marked **PSU** at the back of the *+3*.

The *+3* system is now ready to be switched on.



The standard +3 system set up

Chapter 2
Operating your +3

Subjects covered...

Switching on
Tuning-in your TV
Using the *+3*
The opening menu

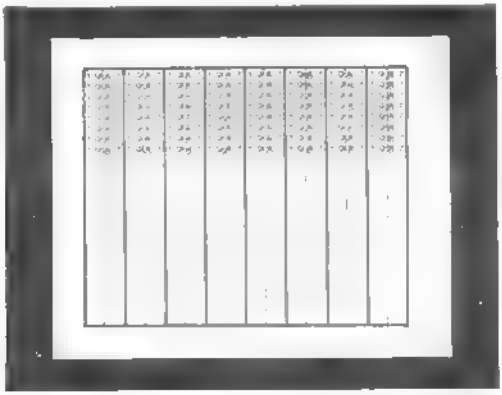
Switching on

Connect the mains plug of the power supply unit to the mains supply socket, and switch on the socket-switch (if necessary). The power indicator lamp on the top panel of the *+3* should illuminate.

Now switch on your TV. On the screen you will probably see either a faint TV picture or just random 'white noise' and hear a loud hissing sound from the TV's speaker. Adjust the TV's volume control until the sound is at a comfortable listening level. The next thing to do is set up the *+3* ready for tuning-in.

Preparing to tune-in your TV

The *+3* is capable of generating its own test signal, enabling you to tune-in the TV accurately. The test signal consists of sixteen vertical colour bars (overprinted with text characters) which appear on the TV screen, and a repeating tone which is reproduced through the TV's speaker. (If you are using a black-and-white TV, then the colour bars appear as varying shades of grey.) You will see and hear the test signal when you have completed the tuning-in of your TV (described ahead).



Switch on the test signal by holding down the **BREAK** key (at the top right of the keyboard) and while it is held down, press and release the **RESET** button (at the left hand side of the ***+3***). Keep the **BREAK** key held down for a few seconds longer, then release it. The test signal will now be generated by the ***+3***, and you should proceed to tune-in your TV as now described.

Push-button TV channel selectors

If your TV ***doesn't*** have push-button channel selectors, then skip to the section ahead entitled 'Manual tuning'.

If your TV ***does*** have push-button channel selectors, then press one of them to select a spare channel (ie. one not normally used for receiving TV or video programmes). Note that if your TV is equipped with an AFC (or AFT) switch, then this should be set to the ***off*** position.

Using the tuning control that corresponds to the selected channel, tune-in to the test signal (shown on the previous page). Make sure that both picture and sound are tuned-in for the best possible results.

When you are satisfied with the tuning, then you may (if your TV is so equipped) set the AFC (or AFT) switch to the ***on*** position.

Finally, adjust the TV's brightness, contrast and colour controls for the clearest display of the text characters within the colour bars.

Now that you have tuned-in one of the TV's push-button channel selectors specifically for the ***+3***, you may thereafter select that particular channel whenever you wish to use the ***+3*** with your TV.

You may now skip to the section ahead entitled 'Using the ***+3***'.

Manual tuning

If your TV isn't equipped with push-button channel selectors, then you will have to use the TV's manual tuning knob to tune-in to your ***+3***.

Having connected and switched on the ***+3*** and TV, switch on the ***+3***'s test signal as described in the previous section entitled 'Preparing to tune-in your TV'.

Tune-in the TV's manual tuning knob until the test signal is received. Make sure that both picture and sound are tuned-in for the best possible results.

Finally, adjust the TV's brightness, contrast and colour controls for the clearest display of the text characters within the colour bars.

Each time that you wish to set up and use the ***+3*** with your TV, you should follow the above manual tuning procedure.

You may now skip to the section ahead entitled 'Using the ***+3***'.

Having problems?

If you have tuned-in your TV satisfactorily, you may now skip to the section ahead entitled 'Using the *+3*'.

If, however, you are unable to tune-in your TV, the following check list may help you to ascertain where the problem lies, and what remedial action you can take.

1. Problem...

The power indicator lamp (on the top panel of the +3) is not illuminated.

Action...

* Check 6-pin DIN plug from power supply unit is plugged into **PSU** socket on computer.
* Check mains plug of PSU is plugged into mains supply socket.
* (If mains supply socket is switched) - Check supply socket switch is on.
* Check connections and fuse in mains plug of PSU.

2. Problem...

The power indicator lamp is illuminated, but no signal whatsoever can be tuned-in on the TV.

Action...

* Check TV is set up and working correctly.
* Check TV is standard UHF type (colour or black-and-white).
* Check aerial lead (supplied) is connected from computer to TV aerial socket.
* (If you have push-button channel selectors) - Check you are tuning-in the channel you selected.

3. Problem...

Only a poor signal from the computer can be tuned-in on the TV.

Action...

* Check TV is set up and working correctly.
* Check aerial lead (supplied) is fully plugged into computer and TV aerial socket.
* (If TV is so equipped) - Check AFC (or AFT) switch is set to off position.
* Check tuning-in has been carried out as accurately as possible.

4. Problem...

A signal from the computer is being tuned-in, but it's not the test signal described above.

Action...

* Check computer's test signal has been switched on (as described in the previous section entitled 'Preparing to tune-in your TV').

5. Problem...

The test signal colour bars appear, but no sound (repeating tone) is audible from the TV's speaker.

Action...

* Check TV's volume control is not at minimum.
* Check tuning-in has been carried out as accurately as possible.

Problem...

The test signal sound (repeating tone) can be heard, but no colour bars can be seen on the TV.

Action...

- Check TV's brightness, contrast and colour controls are not at minimum
- Check tuning-in has been carried out as accurately as possible

7. Problem...

The test signal colour bars and sound are tuned-in, but none of the text characters can be read.

Action...

- Check tuning-in has been carried out as accurately as possible
- Check TV's brightness, contrast and colour controls are adjusted for best results

If you cannot identify the cause of your problem try carrying out the entire procedure (from the beginning of this chapter) again. If the problem still persists, contact your Sinclair dealer

Using the +3

The ***+3*** system should now be fully set up with the test signal colour bars on the screen, and the repeating tone coming from the TV's speaker

We will now switch off the test signal and start using the ***+3*** Press and release the **RESET** button (at the left hand side of the ***+3***) The test signal will disappear from the screen and in its place will be the ***opening menu***.

The opening menu

Note that the opening menu initially indicates which drives are available for use: drive A is the built-in disk drive (at the front of the computer) and drive M is the ***+3***'s internal RAMdisk (more about this in

chapter 8 part 20). If you have connected an external disk drive to the *+3*, then you will see drive B, also indicated on the menu. The opening menu will appear whenever you first plug in and switch on the *+3*, or whenever you press and release the **RESET** button.

As its name suggests, the menu offers you a selection of options. You can choose from one of the four options which appear within the central box on the screen. These are:

`Loader`	- Choose this option if you wish to load Spectrum *+3*, Spectrum *+2* or Spectrum 128 software.
`+3 BASIC`	- Choose this option if you wish to use the *+3* for BASIC programming.
`Calculator`	- Choose this option if you wish to use the *+3* as a calculator only.
`48 BASIC`	- Choose this option if you wish to load Spectrum 48 software from tape (or wish to use the *+3* as a 48K Spectrum).

How to choose an option

Notice that the menu option `Loader` appears to be highlighted by a 'bar'. This means that the `Loader` option is ready to be selected - (the selection hasn't been confirmed yet). For the purpose of this example, let's assume that you don't want to select `Loader`, but that instead, you want to select `+3 BASIC`. This means that you need to move the highlight bar to the option `+3 BASIC`. To do this, use the cursor keys (shown below) until the highlight bar moves to the desired position.

Cursor Keys

When the highlight bar is on `+3 BASIC`, confirm this choice by pressing the **ENTER** key.

The computer then switches to the *+3* BASIC mode. You will see a black horizontal bar (containing the words +3 BASIC) towards the bottom of the screen, and a flashing blue and white blob (called the ***cursor***) at the top left-hand corner.

Don't worry if you know nothing about BASIC - we're not going to do any programming just yet - we'll simply return to the opening menu again. To do this, we use a different menu - this one's called the ***edit menu***. Call up the edit menu by pressing the **EDIT** key.

Again, using the cursor keys and **ENTER**, select the option Exit to return to the opening menu.

You may now select whichever opening menu option you require. Depending upon your selection, refer to the following chapters for further information:

Loader	- Refer to chapters 3 and 4.
+3 BASIC	- Refer to chapters 6 and 8.
Calculator	- Refer to chapter 9.
48 BASIC	- Refer to chapters 4 and 7.

IMPORTANT - Whenever you have finished using the *+3*, always disconnect the power supply unit from the mains supply socket (having first removed any disk from the disk drive).

Chapter 3
How to load disk software

Subjects covered...

Disks for the *+3*
Loading disk software
Abandoning loading

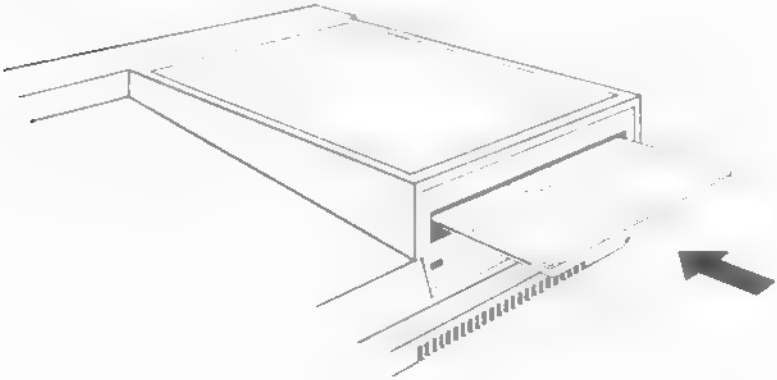
This chapter describes the loading of commercially available disk software

(For a description of the loading saving formatting etc procedures that you would use during BASIC programming see chapter 6 and chapter 8 part 20)

Disks for the +3

The *+3* us[illegible] nd that for [illegible] transfer y[illegible] ther [illegible]

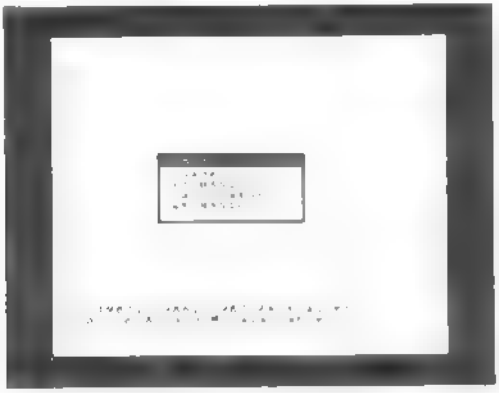
Each side of a disk may be used independently A disk should be inserted with its label facing *outward* from the drive and with the side that you wish to use *face up*



Loading disk software

To load Spectrum *+3*, Spectrum *+2* and Spectrum 128 software (a game, an utility program, etc.) from disk, carry out the following instructions...

1. Set up and switch on the *+3* so that the opening menu appears on the screen...



2. Insert your software disk into the disk drive

3. Press the **ENTER** key ■ select the option `Loader` from the opening menu. (If you don't know about selecting menu options, refer back to chapter 2.)

The software will start to load from disk. On the disk drive, you will see the read/write indicator lamp start to flash on and off (indicating that the disk is being read from). After a few seconds, the screen display will change and the software will be loaded, ready ■ use.

When you have finished using the software and wish ■ use the *+3* for something else, press and release the **RESET** button (at the left-hand side of the *+3*). Always remember that whenever the **RESET** button is pressed, ***everything*** in the computer's memory (RAM) is cleared. You should therefore always make sure that you have completely finished with any program ■ the *+3*'s memory, ***before*** you press the button.

If you are going to switch off the *+3* completely, remember to remove any disk from the disk drive first.

Abandoning loading

If you wish to abandon ■ loading operation, simply press and release the **RESET** button. The *+3* will return to the opening menu.

Chapter 4
How to load tape software

Subjects covered...

Using tape instead of disk
Loading Spectrum *+3*, Spectrum *+2* and Spectrum 128 software
Loading Spectrum 48 software
Abandoning loading

This chapter describes the loading of commercially available pre-recorded tape software

(For a description of the loading, saving, formatting, etc. procedures that you would use during BASIC programming, see chapter 6 and chapter 8 part 20.)

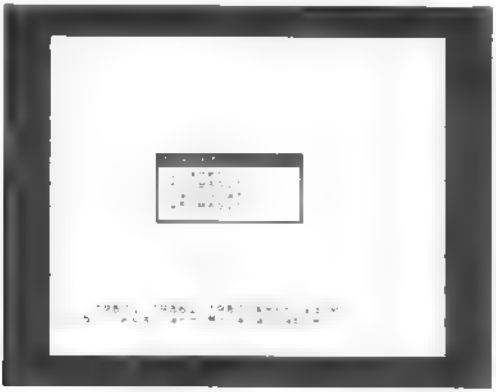
Using tape instead of disk

So far, you have seen how to load software from disk. To use tape, first set up the *+3* system so that your cassette unit is connected to the **TAPE/SOUND** socket ■ the back of the *+3* (details of connections to this socket will be found in chapter 10)

Spectrum +3, Spectrum +2 and Spectrum 128 software

To load Spectrum *+3*, Spectrum *+2* or Spectrum 128 software (a game, an utility program, etc.) from tape, carry out the following instructions

1. Switch on the system so that the opening menu appears on the screen



2. Make sure that no disk is inserted in the disk drive.

3. Press the **ENTER** key to select the option **Loader** from the opening menu. (If you don't know about selecting menu options, refer back to chapter 2.)

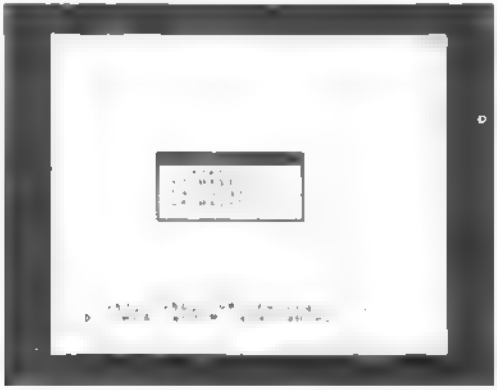
Now skip to the section ahead entitled 'Loading from tape'.

Note that when you select the **Loader** option from the opening menu, the ***+3*** knows that you wish to load from tape (instead of disk) by automatically detecting the absence of a disk in the disk drive. If a disk is inserted, the tape will be ignored.

Spectrum 48 software

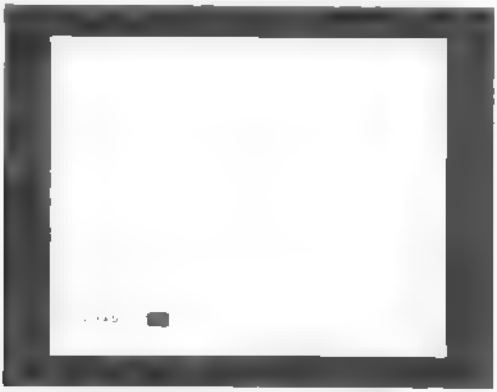
To load Spectrum 48 software (a game, an utility program, etc.) from tape, carry out the following instructions.

1. Switch on the system so that the opening menu appears on the screen.



2. Select the option **48 BASIC** from the opening menu. (If you don't know how to select a menu option, refer back to chapter 2.) The opening menu will disappear and the following message will be displayed at the bottom of the screen.

3. Now press the **J** key once, followed by the " (double quotes) key twice. The screen should look like this.



When you see this message, press **ENTER**.

Now skip to the section ahead entitled 'Loading from tape'.

(If the screen does not correspond to the above picture, then you may have selected the wrong menu option or pressed the wrong key. In this case, press and release the **RESET** button (at the left-hand side of the **+3**) and carry out steps 2 and 3 again.)

Loading from tape

1 Insert the software tape into your cassette unit and make sure that it is rewound to the beginning

2 Play the cassette. As loading commences, the border colour will flash and appear striped, indicating that the program is being 'read' from the tape. If your TV's volume control is turned up, you will also hear a varying high-pitched tone. Again, this is an indication that the program is being read.

Most commercially available software cassettes take a few minutes to load. Initially, the program name may appear (toward the top left-hand corner of the screen) possibly followed by various other displays or messages (these will differ from program to program)

When the program has loaded, stop the cassette. The software is then ready to use

When you have finished using the software and wish to use the ***+3*** for something else, press and release the **RESET** button (at the left-hand side of the ***+3***). Always remember that whenever the **RESET** button is pressed, ***everything*** in the computer's memory (RAM) is cleared. You should therefore always make sure that you have completely finished with any program in the ***+3***'s memory, ***before*** you press the button

Abandoning loading

If, while loading software from tape, you wish to abandon the loading operation, then simply press and release the **RESET** button. The ***+3*** will return to the opening menu

NOTE: Holding the **BREAK** key down while loading Spectrum ***+3***, Spectrum ***+2*** or Spectrum 128 software will return the ***+3*** to the opening menu; holding the key down while loading Spectrum 48 software will return the ***+3*** to the 48 BASIC mode

Chapter 5
The +3 disk drive

Subjects covered...

- Disks and drives
- Insertion
- Write protection
- Read/write indicator lamp
- Eject button

Disks and drives

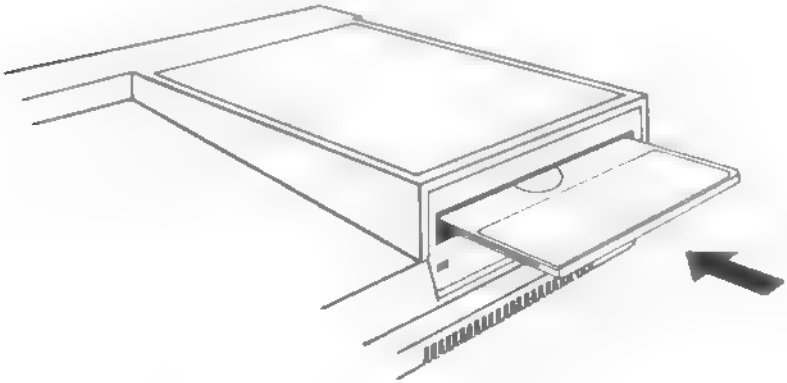
The ***+3*** [illegible]

However [illegible]

If you have connected an additional disk drive to the ***+3*** note that the main disk drive (within the ***+3***) is called ***drive A***: and the second (additional) disk drive is called ***drive B***:

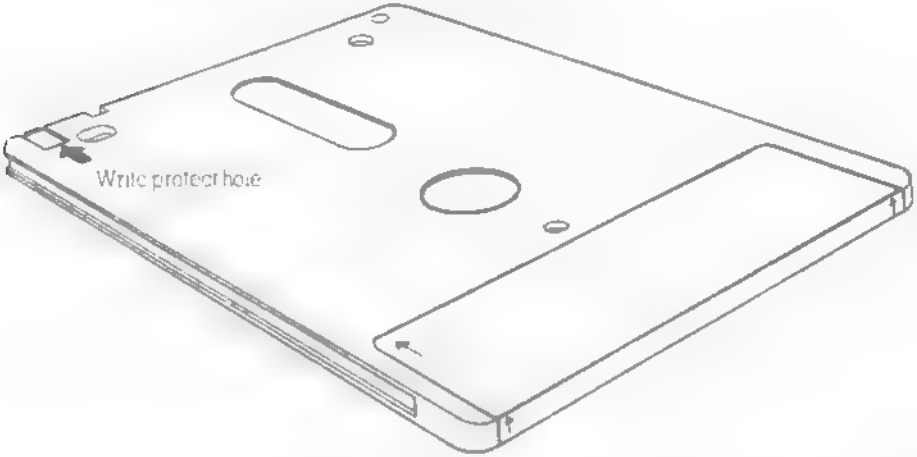
Insertion

Each side of a disk may be used separately. A disk should be inserted with its label facing ***outward*** from the drive, and with the side that you wish to use ***face up***.



Write protection

In the left-hand corner of each side of a blank disk, you will see an arrow pointing to a small shuttered hole. This is called the ***write protect hole***, and allows you to protect the contents of the disk from erasure or 'overwriting'.

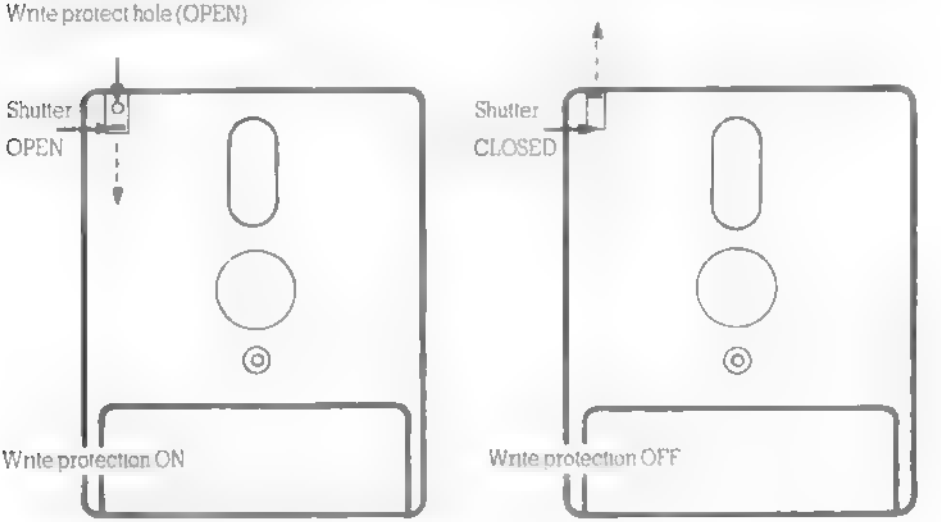


When the hole is ***closed***, data can be written onto the disk by the computer. When the hole is ***open***, however, the disk will ***not*** allow data to be written onto it, thus enabling you to avoid the accidental erasure of valuable programs.

Various disk manufacturers employ [illegible] the write protect hole. The [illegible] disks follows.

To open the write protect hole slide back the small shutter located at the left-hand corner of the disk, and the hole will be opened.

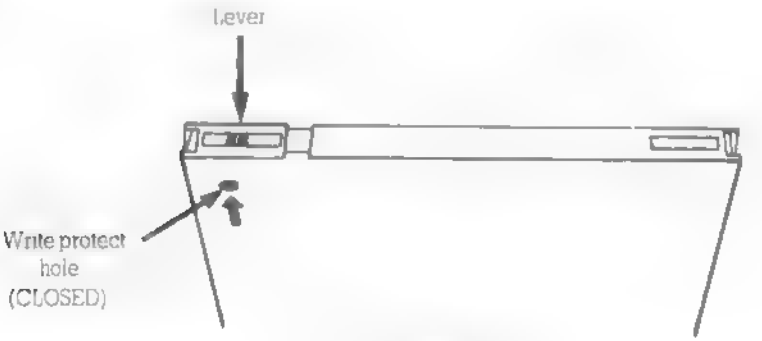
Write protect hole (OPEN)



Once the hole is ***open***, write protection is ***ON***

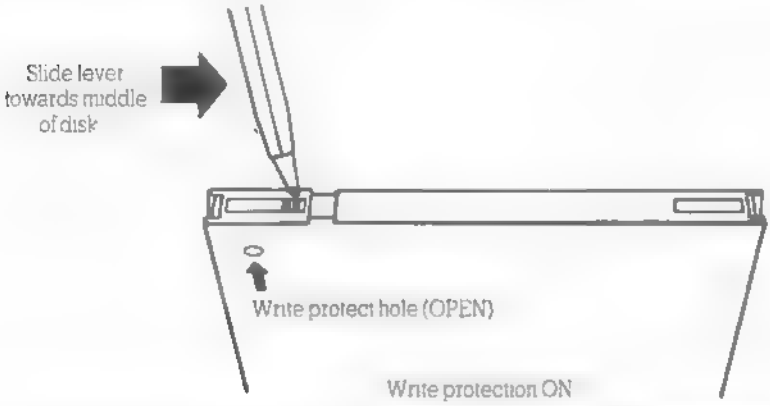
To close the write protect hole, simply slide the shutter to its closed position. Write protection is then ***OFF***

Other manufacturers' disks employ a small lever located in a slot in the left-hand corner



Write protection OFF

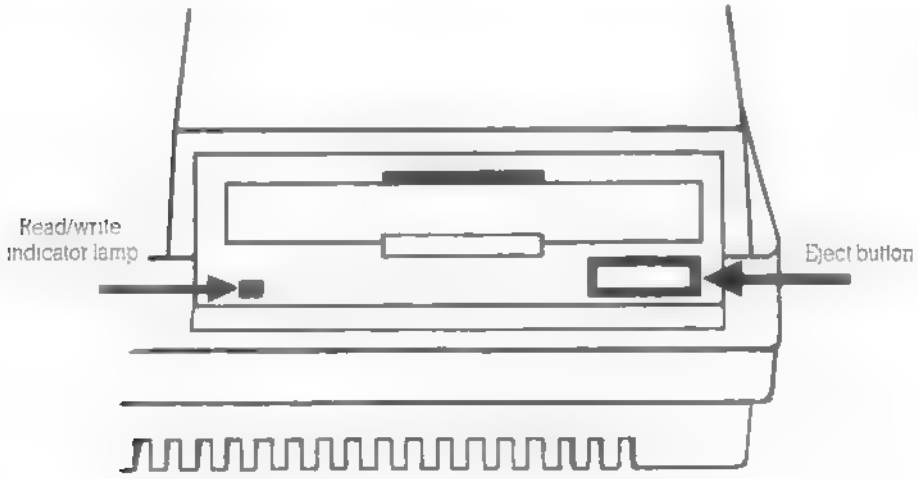
To open the write protect hole on this type of disk: slide the lever towards the middle of the disk (using the tip of a ball-point pen or similar object)



Note that regardless of the method employed to open and close the write protect hole, opening the hole in all cases facilitates protection against overwriting

When your disk is in

At the front of the disk drive you will see a push button (for ejecting the disk), and a red lamp (called the *read/write indicator lamp*)



Read/write indicator lamp

This lamp indicates that data is being read from or written to the disk. Note, however, that if a second disk drive is connected, the read/write indicator lamp on drive B will be constantly on (except when drive A is reading or writing to disk)

Eject button

Pressing in the eject button allows you to remove your disk from the disk drive

Do not press the eject button while the disk is being read from or written to

Always eject your disk from the disk drive before switching the system off

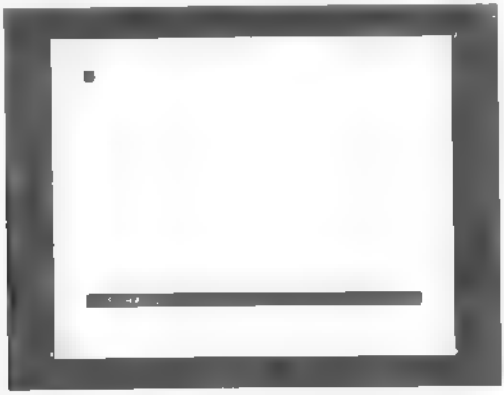
Chapter 6
Introducing +3 BASIC

Subjects covered...

- The editor
- The edit menu
- Renumbering a BASIC program
- Swapping screens
- Listing to the printer
- Typing in a program
- Moving the cursor
- Running a program
- Commands and instructions
- Simple disk operations
- Formatting a disk
- Saving a program
- Filenames
- Disk catalog
- Loading a program
- Error reports

The *+3* has an advanced *editor* to create, modify and run BASIC programs. To enter the editor, select the option **+3 BASIC** from the opening menu, using the cursor keys and **ENTER**. (If you don't know how to select a menu option, refer back to chapter 2.)

The screen should now look like this:



There are three things to notice about this screen:

Firstly, there is a flashing blue and white blob in the top left-hand corner. This is called the ***cursor***, and if you type any letters at the keyboard, then they will appear on the screen at the position of the cursor.

Secondly, there's a black bar towards the bottom of the screen. This is called the ***footer bar***, and tells you which part of the ***+3***'s built-in software you're using. At the moment, it says **+3 BASIC** because that's the name of the editor.

The last item of note at the moment is the small screen. This fits between the footer bar and the bottom of the screen, and is currently blank. It only has room for two lines of text, and is most often used by the ***+3*** when it detects an error and needs to print a ***report*** to say so. It does have other uses, however, and these will be described later.

Now press the **EDIT** key. You will notice two things happen - the cursor vanishes, and a new menu appears. This is called the ***edit menu***.

The edit menu's options are selected in the same way as for the opening menu (by using the cursor keys and **ENTER**).

Taking the options in turn:

+3 BASIC - This option simply cancels the edit menu and restores the cursor. On the face of it - not very useful, however, if **EDIT** is pressed accidentally, then this option allows you to return to your program with no damage done.

Renumber - BASIC programs use ***line numbers*** to determine the order of the instructions to be carried out. You enter these numbers (which can be any whole-number from 1 to 9999) at the beginning of each program line you type in. Selecting the **Renumber** option causes the BASIC program's line numbers to start at line 10 and go up in steps of 10. BASIC commands which include references to line numbers (such as **GO TO**, **GO SUB**, **LINE**, **RESTORE**, **RUN** and **LIST**) also have these references renumbered accordingly.

If for any reason it's not possible to renumber, perhaps because there's no program in the +3, or because Renumber would generate line numbers greater than 9999, then the +3 makes a low-pitched bleep and the menu goes away

A useful aid to this renumbering facility can be found in chapter 8 part 33

Screen - This option moves the cursor into the smaller (bottom) part of the screen, and allows BASIC to be entered and edited there. This is most useful for working with graphics, as any editing in the bottom screen does not disturb the top screen. To switch back to the top screen (which you can do at any time whilst editing), select the edit menu option Screen again.

Print - If a printer is connected, this option will print-out a listing of the current program to it. When the listing has finished, the menu will go away and the cursor will come back. If for some reason the computer cannot print (eg. the printer is not connected or is off-line), then pressing the **BREAK** key twice will return you to the editor

Exit - This option returns you to the opening menu - the +3 retains any program that you were working on in the memory. If you wish to go back to the program again, select the option +3 BASIC from the opening menu

If you select the opening menu option 48 BASIC (or if you switch off or reset the +3), then any program in the memory will be lost. (You may, however, use the opening menu option Calculator without losing a program in the memory.)

Reset the computer and select +3 BASIC. Now type in the line below. As you type it in, the characters will appear on the screen (a character is a letter, number, space, etc.). Note that to type in the equals sign = you should hold down the **SYMB SHIFT** key, then press the L key once. Try typing in the line now

```
10 for f=1 to 100 step 10
```

then press **ENTER**. Providing you have spelt everything correctly the +3 should have reprinted the line with the words FOR TO and STEP in capital letters, like this..

```
10 FOR f=1 TO 100 STEP 10
```

The +3 should have also emitted a short high-pitched bleep, and moved the cursor to the start of the next line

If the line remains in small letters and you hear a low-pitched bleep, then this indicates that you have typed in something wrong. Note also that the colour of the cursor changes to red when a mistake is detected, and you must correct the line before it will be accepted by the +3. To do this, use the cursor keys to move to the part of the line that you wish to correct, then type in any characters you wish to insert (or use the **DELETE** key to remove any characters you wish to get rid of). When you have finally corrected the line, press **ENTER**

Now type in the line below...

(The colon : is obtained by **SYMB SHIFT** and **Z**, and the minus sign - is obtained by **SYMB SHIFT** and **J**.)

```
20 plot 0,0:draw f,175:plot 255,0:draw -f,175
```

..then press **ENTER**. On the screen you will see..

```
10 FOR f=1 TO 100 STEP 10
20 PLOT 0,0: DRAW f,175: PLOT
   255,0: DRAW -f,175
```

Don't worry about line 20 'spilling over' onto the next line of the screen - the computer will take care of this and align the text so that it is easier to read. Unlike a typewriter, there's no need for you to do anything when you approach the end of a screen line because the ***+3*** detects this automatically and moves the cursor to the beginning of a new line.

The final line of this program to type in is

```
30 next f
```

..again, press **ENTER**

The numbers at the beginning of each line are called ***line numbers*** and are used to identify each line. The line you just typed in is line 30, and the cursor should be positioned just below it. As an exercise, we will now edit line 10 (to change the number **100** to **255**). Press the cursor up ↑ key (four times) until the cursor has moved up to line 10. Now press the cursor right ↓ key until the cursor has moved to the right of **100**. Press **DELETE** three times and you will see the **100** disappear. Now type in **255** and press **ENTER**. Line 10 of the program has now been edited.

```
10 FOR f=1 TO 255 STEP 10
```

The computer has opened up a new line in preparation for some new text. Type.

```
run
```

Press **ENTER** and watch what happens. Firstly, the footer bar and the program lines are cleared off the screen as the ***+3*** BASIC editor prepares to hand over control to the program you've just typed in. Then the program starts, draws a pattern, and stops with the report

```
0 OK, 30:1
```

Don't worry about what this report means.

Press **ENTER**. The screen will clear and the footer bar will come back, as will the program listing. This takes about a second or so, during which time the ***+3 won't*** be taking input from the keyboard, so don't try and type anything while it's all happening.

You've just done most of the major operations necessary to program and use a computer! First, you've given the ***+3*** a list of instructions. ***Instructions*** tell the ***+3*** what to do (like the instruction `30 NEXT f`). Instructions have a line number and are 'stored away' rather than used immediately you type them in. Then you gave the ***+3*** the command `run` to execute the stored program.

Commands are just like instructions, only they ***don't*** have line numbers and the ***+3*** carries them out immediately (as soon as **ENTER** is pressed). In general, any instruction can be used as a command, and vice versa - it all depends on the circumstances. Every instruction or command must have at least one ***keyword***. Keywords make up the vocabulary of the computer, and many of them require parameters. In the command `DRAW 40,200` for example, `DRAW` is the keyword, while `40` and `200` are the parameters (telling the computer exactly ***where*** to do the drawing). Everything the computer does in BASIC will follow these rules.

Now press **EDIT** and select the `Screen` option. The editor moves the program down into the bottom screen, and gets rid of the footer bar. You can only see line 10 of the program as the rest is 'hiding' off-screen (you can prove this by moving the cursor up and down).

Press **ENTER** then type..

```
run
```

Press **ENTER** again, and the program will run exactly the same as before. But this time, if you press **ENTER** afterwards, the screen doesn't clear, and you can move up and down the program listing (using the cursor keys) without disturbing the top screen. If you press **EDIT** to get the edit menu, you might think that this would mess up the top screen. However, the ***+3*** remembers whatever's behind the edit menu and restores it when the menu is removed.

To prove that the editor really is working in the bottom screen, press **ENTER** and change line 10 to..

```
10 FOR f=1 TO 255 STEP 7
```

...by moving the cursor to the end of line 10 (just to the right of `STEP 10`), then pressing **DELETE** twice, and typing 7 (press **ENTER**).

Now type...

```
go to 10
```

(Press **ENTER**.) The keywords `go to` tell the ***+3*** not to clear the screen before starting the program. The modified program draws a slightly different pattern on top of the old one. You may continue editing the program ■ add further patterns, if you wish.

A word of warning - while editing in the bottom screen, ***don't*** try to edit instructions which are more than two screen lines long. Otherwise, when the editor comes across an instruction which has its beginning or its end off-screen, it may become 'confused'. (The same is true of the top screen, but of course, this is unlikely to cause any problems as the screen is so much larger.)

One thing you may notice while you're typing away is that **CAPS SHIFT** and the number keys used together do strange things. **CAPS SHIFT** with **5**, **6**, **7** and **8** move the cursor about. **CAPS SHIFT** with **1** calls up the edit menu. **CAPS SHIFT** with **0** deletes a character. **CAPS SHIFT** with **2** is equivalent to **CAPS LOCK**, and finally **CAPS SHIFT** with **9** selects graphics mode. All of these functions are available using the dedicated keys on the ***+3*** and so there is no reason why you should ever want to use the above **CAPS SHIFT** and number key alternatives.

Simple disk operations

You have now seen how to place a program into the computer's memory by typing it in. This is all very well the first time you write a particular program, but what about if you switch off the computer and want to use the same program the next day? Surely you don't have to type it [illegible] in again from scratch -the answer, of course, is no - the disk drive section of the ***+3*** allows you to ***save*** a program from the computer's memory onto a disk, and to ***load*** a program from a disk into the computer's memory. This means that you can type in a program, save it to disk, then happily switch off the ***+3*** knowing that next time you switch it on, you'll be able to load that same program back into the memory.

The final part of this chapter, therefore, deals with these two very important operations (saving and loading). However, ***before*** you can do either of these, you will be shown how [illegible] prepare a brand new disk so that it is ready for saving programs onto. This preparation process is called ***formatting***.

You will need to have a brand new blank disk [illegible] hand as you work through this chapter. **Whatever you do - don't use a disk with any valuable software, games, etc. on it.**

Disks and tapes

Even if you are familiar with tape saving and loading, it is worth pointing out two important points which must be remembered when dealing with disks.

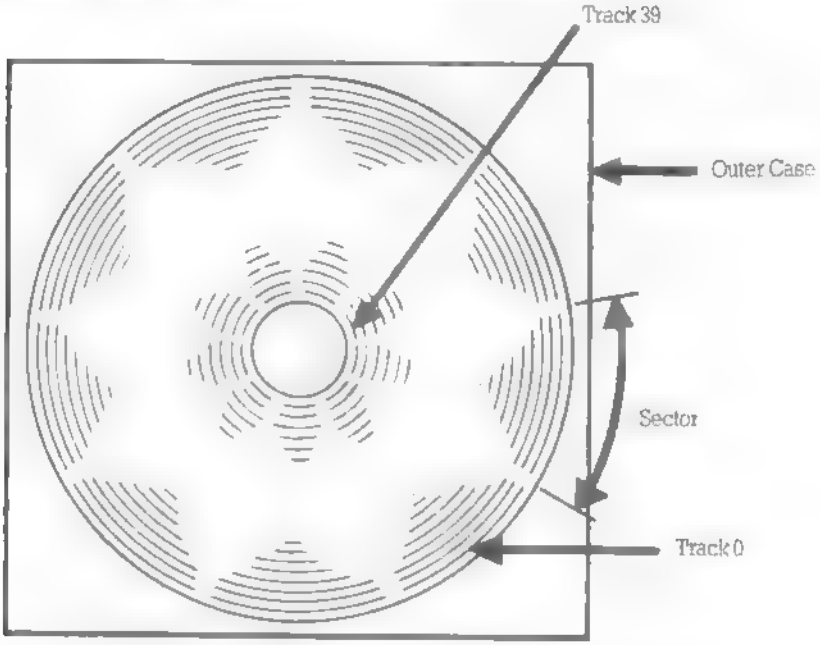
Firstly, a brand new blank disk ***cannot*** be simply taken out of its wrapper and recorded onto (as [illegible] the case with a tape) - instead, each side of a disk must be formatted first. **Note that the formatting process completely erases that side of the disk.**

Secondly, it is important that disk files are correctly 'named'. Filenames on tape may vary greatly in length and may at times be omitted. Not so with disks - disk filenames must conform to very strict standards (and you will read about these shortly in the section ahead entitled 'Filenames').

Formatting a disk

Formatting can be likened to building a series of shelves and pigeonholes on a disk prior to the storage of information on those shelves. In other words, formatting lays down an organised framework around which data can be put in or taken out.

The formatting process divides the disk into 360 distinctly separate areas.



There are 40 concentric tracks from the outside of the disk (Track 0) ■ the inside (Track 39), and each track ■ divided into 9 sectors

Each portion of track in a sector can store up to 512 bytes of data, hence the total available space on each side of a disk is 180 kilobytes (180K). Note that 7K of the 180K is reserved for the computer's own use, this leaves 173K per side for your programs.

We will now format a new blank disk, and save the program below...

```
10 FOR f=1 TO 255 STEP 7
20 PLOT 0,0: DRAW f,175: PLOT
   255,0: DRAW -f,175
30 NEXT f
```

..which should still be in the memory from the previous exercise (check that the above program is currently in the memory by pressing **ENTER** then typing..

```
List
```

(Press **ENTER** again.) If the program isn't in the memory (or you have since switched off the ***+3***), then switch it on, select **+3 BASIC** and type in the above program)

Insert side 1 of a new blank disk into the disk drive and type...

```
format "a:"
```

(Press **ENTER**.) The read/write indicator lamp on the disk drive will start to flash on and off. About 30 seconds later, you will see the report...

```
0 OK, 0:1
```

You have now formatted side 1 of the disk. Once you have done this, you should not need to format side 1 of that disk ever again.

(If you don't receive the above report (and some other message appears instead), check the section entitled 'Error reports' at the end of this chapter.)

Saving a program

Having formatted side 1, it is now ready for saving programs onto.

In order that each program file on a disk can be identified, you ***must*** give the program a ***filename*** when you save it. For example, as the program that you are about to save draws a patterned picture, save the program using the name 'pattern.pic', ie. type in...

```
save "pattern.pic"
```

(Press **ENTER**.) After a few seconds, you will see the report...

```
0 OK, 0:1
```

The program is now saved onto disk.

(If you don't receive the above report (and some other message appears instead), check the section entitled 'Error reports' at the end of this chapter.)

Filenames

Note that a filename on disk consists of two parts (***fields***). The first field is obligatory and can contain up to 8 characters (letters and numbers may be used but no spaces or punctuation marks). In the above example filename, 'pattern' is the first field.

The second field is optional. You can use up to 3 characters (but again no spaces or punctuation). In the above example filename, 'pic' is the second field.

If you use two fields in a filename, they must be separated by a dot (eg. 'pattern.pic').

Disk catalog

A catalog of the disk (in alphabetical order) can be displayed by typing in :

```
cat
```

(Press **ENTER**) The filenames of all the programs on that side of the disk will be displayed, together with each file's length (to the nearest higher kilobyte) The amount of free space will also be indicated.

```
PATTERN .PIC    1K
172K free
```

Loading a program

Imagine that you have switched off the ***+3*** and later want to load the program you have just saved. Do this now by resetting the ***+3*** (using the **RESET** button) and selecting the option **+3 BASIC** from the opening menu Type in

```
load "pattern.pic"
```

(Press **ENTER**) After a few seconds, you will see the report...

```
0 OK, 0:1
```

The program is now loaded from disk Press **ENTER** and you will see the program listing displayed.

(If you don't receive the above report (and some other message appears instead), check the next section entitled 'Error reports')

Once loaded, you may run the program by simply typing.

```
run
```

and pressing **ENTER**, as before

Error reports

If you don't correctly carry out the instructions in this section, you may receive various ***error reports***. If so, identify the report (from those shown below), read the explanation given, and then take the necessary corrective action.

```
Drive not ready
```

The above report means that you have probably forgotten to insert a disk into the disk drive. If there is a disk inserted in the disk drive, then eject it, re-insert it and try again.

```
Disk is write protected
```

The above report means that you are trying to format, or save a program to, a disk which has its write protect hole open. Eject the disk, close its write protect hole, re-insert the disk and try again

```
File not found
```

The above report means that you are trying to load a program which doesn't exist on that side of the disk. Eject the disk, make sure that the correct disk is inserted (the right way up) and try again. Take care to ensure that you accurately type in the filename to load.

```
Bad filename
```

or.

```
Invalid filename
```

The above reports mean that you are trying to load or save a program using an illegal filename (or no filename at all). Read the section entitled 'Filenames' earlier in this chapter, and try again

```
Disk is already formatted
A to abandon, other key continue
```

The above report means that you are trying to format a disk that has already been formatted. In general, a disk should need formatting only once (at the beginning of its life). In rare cases, a disk may become corrupted and there will be no alternative other than to format it again. However, unless this is the case, you should always type **A** (to abandon) when you see the above report

NOTE - If you don't type **A**, then the formatting process will go ahead and ***completely erase*** that side of the disk (as soon as you press a key)

If you find that one particular disk (or side of a disk) keeps requiring formatting, then it is likely that the disk itself is damaged and you should avoid using it in future

Some commands that fail will produce reports that offer you the options .

```
- Retry, Ignore or Cancel?
```

If you receive the above options, then:

. typing **R** (after taking the necessary corrective action) makes the computer retry the command.

..typing **I** makes the computer ignore the reason that the command failed in the first place and continue regardlessly (typing **I** is therefore not recommended unless you know exactly what you're doing).

typing **C** abandons the command (this may be followed by the appearance of another report)

Further information

Further information on disk operations (together with details of how to use the +3's RAMdisk and how to use an external cassette unit) can be found in chapter 8 part 20. A guide to +3DOS (the +3 Disk Operating System) will be found in chapter 8 part 27.

Chapter 7
Using 48 BASIC

Subjects covered...

Using the *+3* as a 48K Spectrum
Entering 48 BASIC mode
The keyboard under 48 BASIC
Program entry
Editing the current line

The *+3* has the ability to act exactly like a 48K Spectrum (or Spectrum +). This is achieved by selecting the option **48 BASIC** from the opening menu. In 48 BASIC mode, many of the enhanced features of the *+3* (such as the disk drive, extra memory, full screen editor, multi-channel sound, **RS232/MIDI/AUX** interfaces and RAMdisk) cannot be used. The **JOYSTICK 1** and **JOYSTICK 2** sockets will still operate, however.

The 48 BASIC mode is included for compatibility reasons only - there is no advantage in using 48 BASIC mode (instead of *+3* BASIC mode) to write programs, and it is ***not*** recommended. The following information is included for reference only, or for anybody who is used to the old 48K Spectrum and wants to use the machine immediately without having to learn about the *+3* BASIC editor.

There are, in fact, two methods of entering the 48 BASIC mode: the first is by selecting the **48 BASIC** option from the opening menu (if you don't know how to select a menu option, refer back to chapter 2). When 48 BASIC starts up, you will see the following on the screen...

The second method allows you to enter the 48 BASIC mode while editing a *+3* BASIC program. To do this (while in *+3* BASIC mode), type...

.. and press **ENTER**. The *+3* will respond with an OK message and will have changed to 48 BASIC mode, retaining any program that you had in memory. Once in 48 BASIC mode, there is no way back to *+3* BASIC mode apart from resetting the *+3* (or switching off, then on again).

One major difference between 48 BASIC and *+3* BASIC is in the entering and editing of programs. (Note also that in 48 BASIC the tokens SPECTRUM and PLAY have replaced the user defined graphics characters for the keys **T** and **U** under *+3* BASIC (values 163 and 164).)

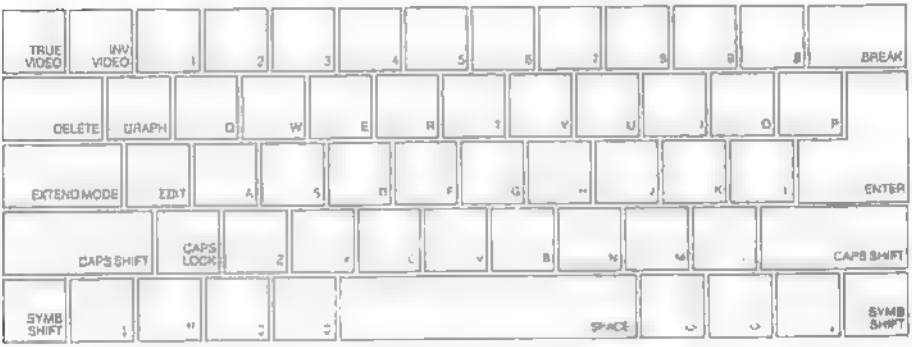
Once in BASIC mode, the keyboard performs as follows.

All the BASIC commands, functions and operators are available directly from the keyboard rather than needing to be spelled out. In order to accommodate all these functions and commands, some keys have five or more distinct meanings, obtained partly by 'shifting' the keys (ie. pressing either **CAPS SHIFT** or **SYMB SHIFT** together with the required key); and partly by having the machine in different modes. The flashing cursor contains a letter (K, L, C, E or G) to indicate which mode you are operating in.

K (for Keywords) mode automatically replaces L (for Letters) mode when the machine is expecting a command or program line (rather than input data), and from its position on the line the *+3* knows that it should expect either a line number or a keyword. K mode occurs at the beginning of a line, or after a colon : (except in a string), or after the keyword THEN. Whenever the K cursor appears, the next key pressed will be interpreted as either a keyword or a line number, as follows..

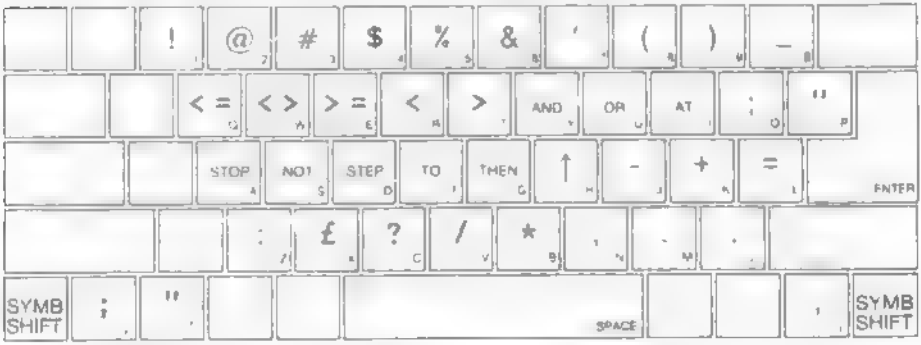
The keyboard in K mode

L (for Letters) mode normally occurs at all times (other than K mode, described above). Whenever the L cursor appears, the next key pressed will be interpreted as per the legends on the key-tops themselves, ie.



The keyboard in L mode

In both K and L modes, pressing **SYMB SHIFT** together with a key will be interpreted as follows..

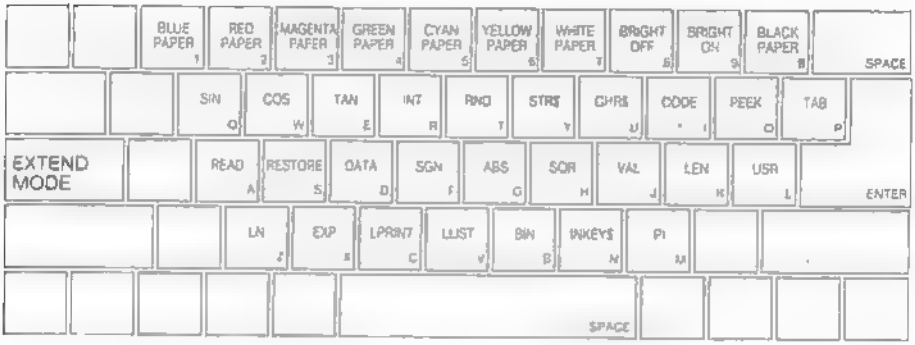


The keyboard using **SYMB SHIFT** in K or L mode

Using **CAPS SHIFT** in L mode simply converts small letters to capitals. In K mode, however, **CAPS SHIFT** does not affect the keywords.

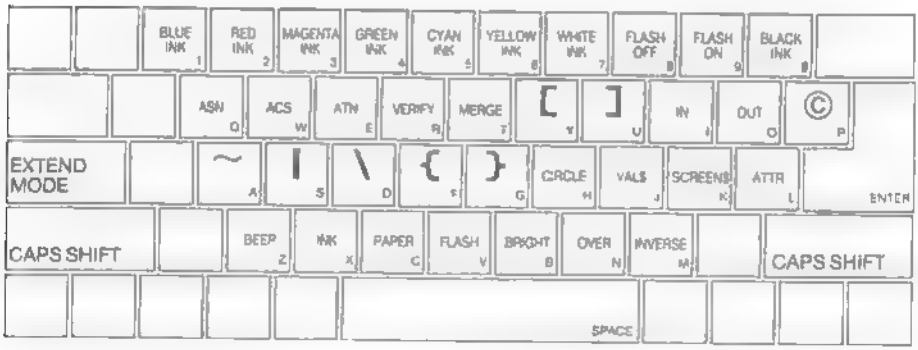
C (for Capitals) mode is a variant of L mode whereby all letters appear as capitals. The **CAPS LOCK** key is used to change from L mode to C mode, and back again.

E (for Extended) mode is used to obtain further characters, mostly tokens. It is entered by pressing the **EXTEND MODE** key, and lasts for only one character (or key depression) thereafter. Whenever the **E** cursor appears, the next key pressed will be interpreted as follows...



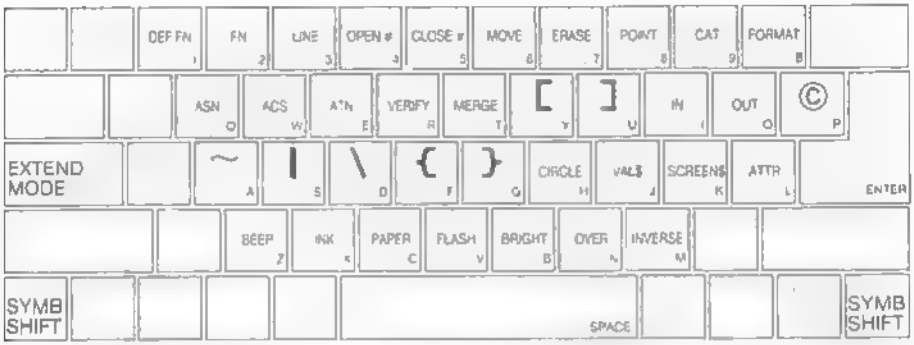
The keyboard in **E** mode

Applying **CAPS SHIFT** while in **E** mode, the next key pressed will be interpreted as follows...



The keyboard using **CAPS SHIFT** in **E** mode

Applying **SYMB SHIFT** while in E mode, the next key pressed will be interpreted as follows...

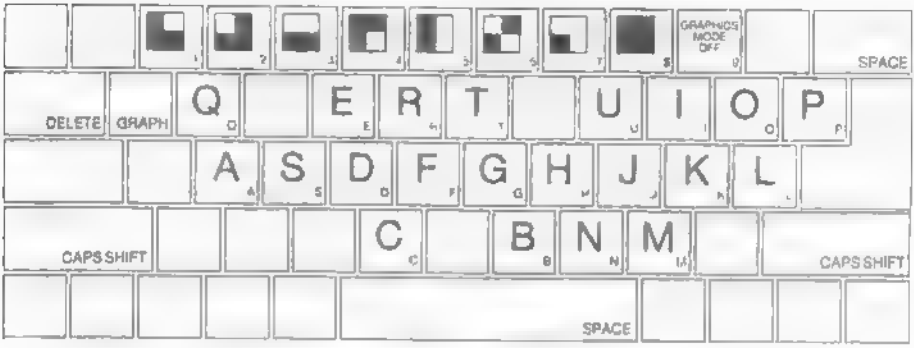


The keyboard using **SYMB SHIFT** in E mode

G (for Graphics) mode occurs when **GRAPH** is pressed, and lasts until it is pressed again (or **9** is pressed on its own). A number key will give a mosaic graphic, and each of the letter keys (apart from **V**, **W**, **X**, **Y** and **Z**) will give a user-defined graphic which, until it is defined, will look identical to a capital letter. Whenever the G cursor appears, the next key pressed will be interpreted as follows...

The keyboard in G mode

Applying **CAPS SHIFT** while in G mode *inverts* the mosaic graphics (ie. the ink colour becomes the paper colour, and the paper becomes the ink colour). Hence, the next key pressed will be interpreted as follows..



The keyboard using **CAPS SHIFT** in G mode

General keyboard notes

If any key is held down for more than 2 or 3 seconds, it will start ***repeating***. Keyboard input appears in the bottom half of the screen as it is typed, each character (single symbol or compound token) being inserted just before the cursor. The cursor can be moved left and right using the cursor control keys ⇦⇨ (to the left of the space bar). The character to the left of the cursor can be removed using **DELETE**.

When **ENTER** is pressed, the line is either executed, entered into the program, or used as input data. If the line contains a ***syntax error***, however, a flashing question mark ? appears next to the error.

As program lines are entered, a listing is displayed in the top half of the screen. The last line entered is called the current line and is indicated by the symbol > after the line number. Any line in the program may be selected as the current line (for editing purposes) by using the up and down cursor keys ⇧⇩ (to the right of the space bar). To then edit the selected current line, press the **EDIT** key. (Editing takes place at the bottom of the screen.)

When a command is executed or a program is run, output is displayed in the top half of the screen and remains there until either **ENTER** or the cursor up or down key ⇧⇩ is pressed. At the bottom of the screen appears a ***report*** giving a code (digit or letter) referred to in part 29 of chapter 8. This report remains on the screen until a key is pressed and the ***+3*** returns to K mode.

Chapter 8
The +3 BASIC programmer's guide

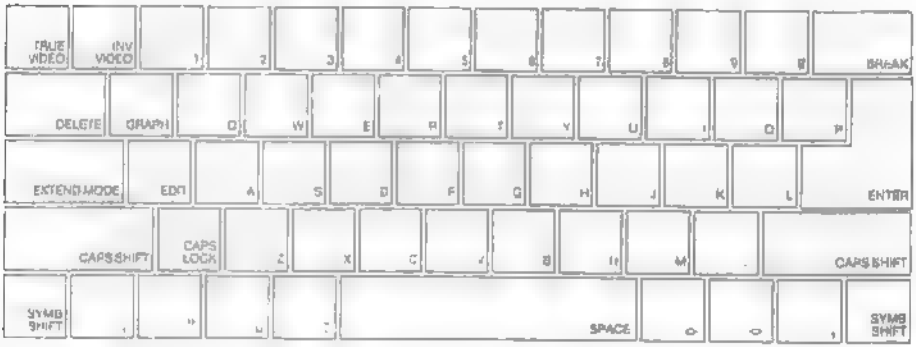
Part 1
Introduction

Whether you read chapter 6 first, or came straight here, you should be aware that...

> *Commands* are obeyed straight away.
>
> *Instructions* begin with a line number and are stored away for later use.

This guide to BASIC starts by repeating some of the information given in chapter 6 (Introducing *+3* BASIC), but in greater detail. You may also find exercises at the end of some sections - don't ignore these, as many of them illustrate points that are hinted at in the text. Look through them, and do any that interest you or that seem to cover ground that you don't understand properly.

The Keyboard



The characters used on the *+3* comprise not only single symbols (letters, digits, etc.) but also compound tokens (keywords, function names, etc.). Everything must be typed in full, and in most cases it doesn't matter whether capital letters (known as *UPPER CASE*) or small letters (*lower case*) are used. There are three sorts of keys on the keyboard: letter and number keys (called alphanumeric keys), symbol keys (punctuation marks), and control keys (things like **CAPS SHIFT**, **DELETE** and so on).

The most commonly used keys for BASIC are the alphanumeric keys. When a letter key is pressed, a lower case letter will appear on the screen together with a flashing blue and white blob called the *cursor*. To get an upper case letter, the **CAPS SHIFT** key should be held down while the letter is typed.

If you wish to continuously type upper case letters, then pressing the **CAPS LOCK** key once will make all subsequent letters typed upper case. To return to lower case letters, simply press **CAPS LOCK** again.

To type the symbols which appear on the alphanumeric keys on the keyboard, ie.

! @ # $ % & ' ( ) _ < > ↑ – + = ; £ ? / *

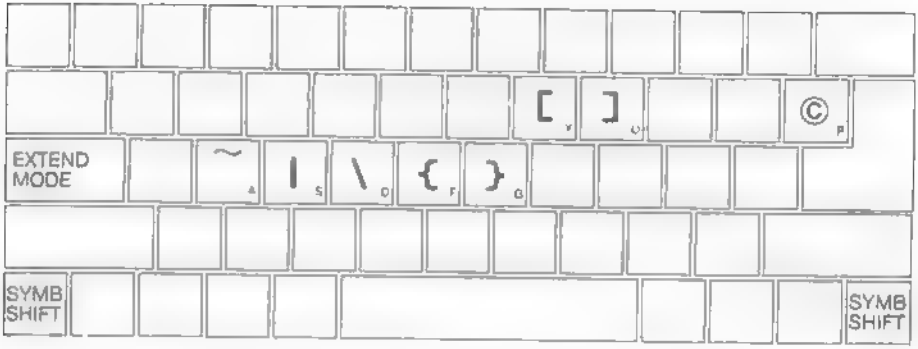
simply hold down the **SYMB SHIFT** key while the alphanumeric key with the required symbol on it is pressed (see the following diagram).

Symbols available using **SYMB SHIFT**

Additionally, the symbols

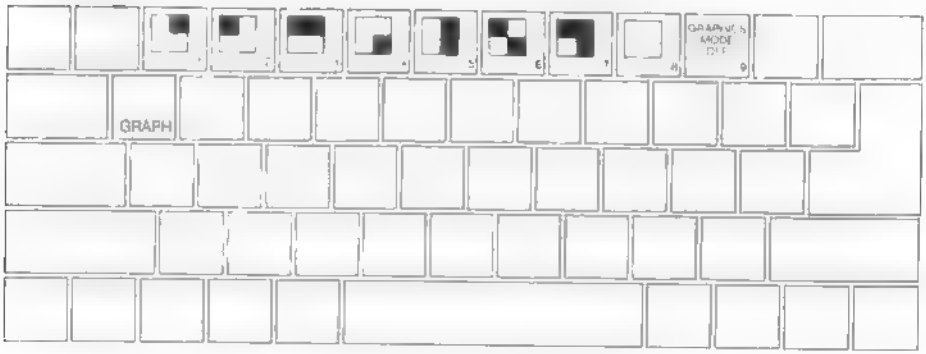
[ ] © ¬ | \ { }

can be obtained by first pressing the **EXTEND MODE** key once, then holding down **SYMB SHIFT** while pressing the appropriate alphanumeric key (see the following diagram).



Symbols available using **SYMB SHIFT** in **EXTEND MODE**

To enter graphics mode, the **GRAPH** key is pressed once. Mosaic graphics (see the following diagram) can then be produced by pressing the number keys (except **9** and **0**). Pressing the letter keys (except **T**, **U**, **V**, **W**, **X**, **Y** and **Z**) produce user-defined graphics (if set up).



Mosaic graphics available using **GRAPH**

To obtain inverted mosaic graphics, press the above number keys while holding down **CAPS SHIFT**

General keyboard notes

If any key is held down for more than 2 or 3 seconds, it will start repeating. As keys are pressed, a line will be built up on the screen. A line, by the way, means a line of BASIC, and may easily be several lines long on the screen. The cursor keys ⇦⇨⇩⇧ can be used to move about the line, and if the part of the line that the cursor is moved to is off screen, then the text on screen will scroll up or down to display it. Any characters typed will be inserted at the cursor, and pressing **DELETE** causes the character to the left of the cursor to be removed. As soon as **ENTER** is pressed or any attempt is made to move the cursor off the line, the ***+3*** checks to see if the line makes sense. If it does, then there is a high-pitched bleep, and the line is either acted upon immediately or stored away as part of a program. If the line contains an error, then the ***+3*** generates a low-pitched bleep and moves the cursor to the area where it thinks the error is (the colour of the cursor also changes to red to indicate the error). It is impossible to move off a line which contains an error - the ***+3*** will always move the cursor back.

The monitor screen

This has [illegible] lines (each being 32 characters long) and is divided into two parts. The larger (top) part of the screen is at most 22 lines and displays either a listing or program output. It is the one used most often for editing. When printing in the top part has reached its bottom limit, the contents scroll up by one line. If, however, scrolling would mean losing a line that you haven't yet had a chance to see, then the ***+3*** stops with the message..

```
scroll?
```

Pressing any key (except **N**, **BREAK** or the space bar) will let scrolling continue.

Pressing one of the keys **N**, **BREAK** or the space bar will make the program stop with the report

```
D BREAK - CONT repeats
```

The smaller (bottom) part of the screen is used for editing short programs, entering input data, entering direct commands (where the main screen must not be used, eg. graphics programs), and also for displaying reports.

Program entry

If the program being entered gets bigger than the screen size, then the ***+3*** attempts to display the area of most interest (usually the last line entered together with its surrounding lines). You may, however, specify a different area of the program to be displayed using the command

```
LIST xxx
```

..where 'xxx' is a line number, telling the **+3** to bring a specified area of the program into view.

When a command is executed or a program is run, output is displayed in the top part of the screen and remains there when the program finishes (until a key is pressed). If the program is being edited in the bottom part of the screen, then any output in the top screen will stay there until it is either overwritten, scrolled off, or a `CLS` command is issued. The bottom screen may display a ***report*** giving a code (digit or letter) referred to in part 29 of this chapter. This report remains in the bottom screen until a key is pressed.

While the **+3** is running a BASIC program, the **BREAK** key is checked every so often. This happens at the end of a statement, during use of the cassette unit (if connected) or printer (if connected), or while music is being played. If the **+3** finds that the **BREAK** key is pressed, then program execution stops and displays a report. The program may then be edited.

Part 2
Simple programming concepts

Subjects covered...

Programs
Line numbers
Editing programs using ⟨↑⟩ ⟨←⟩ ⟨→⟩
RUN, LIST
GO TO, CONTINUE, INPUT, NEW, REM
PRINT
Stopping a program

Type in the following first two lines of a program (which will eventually print the sum of two numbers) Don't forget to press **ENTER** after you type each line..

```
20 print a
10 let a=10
```

Note that the screen looks like this.

```
10 LET a=10

20 PRINT a
```

As we have already discussed - because these lines began with numbers, they were not obeyed immediately but were stored away as program lines. You will have also noticed here that the line numbers govern the order in which the program lines are to be executed, and as you can see on the screen, the *+3* sorts all the lines into order whenever a new line is entered.

Note also that although we typed each line in lower case letters, the keywords (ie. PRINT and LET) were converted to upper case as soon the line was entered and accepted by the *+3*. From now on, we will show keywords to be typed in upper case letters; however, you may continue to type in lower case letters.

(By the way, if you don't know what a keyword is you should have studied chapter 6 before reading this chapter.)

So far you have only entered one number, so type

```
15 LET b=15
```

...and press **ENTER**. Now you need to change line 20 to...

```
20 PRINT a+b
```

You could type out the replacement line in full, but it is far easier to move the cursor (using the cursor keys) to just after the **a**, and then type.

```
+b                    (don't press ENTER yet)
```

Check that the line then reads.

```
20 PRINT a+b
```

...then press **ENTER**. The cursor will move to the line below, and the screen should look like this

```
10 LET a=10
15 LET b=15
20 PRINT a+b
```

What you have done in this program is to have ***assigned*** the value [illegible] to the ***variable*** called **a**, and the value 15 to the variable called **b**. You have then instructed the computer to print the sum of these two values by simply adding the two variables.

Run this program by typing.

```
RUN
```

...and pressing **ENTER**. The sum of the two numbers will be displayed.

```
25
```

Run the program again and then afterwards, press **ENTER** and type.

```
PRINT a,b
```

Now press **ENTER** again and notice how the values of the variables **a** and **b** are still in the ***+3***'s memory, even though the program has finished.

```
10          15
```

Mistakes

If you enter a line by mistake, say...

```
12 LET b=8
```

...and you wish to delete the line, then simply type...

```
12
```

...and press **ENTER**. Line 12 will vanish, and the cursor will reappear where line 12 used to be.

Now type

```
30
```

...and press **ENTER**. The ***+3*** will search for line 30, and since there isn't one, it will fall off the end of the program. The cursor will be positioned just after the last line. If you enter any non-existent line number (such as 30) then the ***+3*** will place the cursor where it thinks the line would have been if it really existed. This can be a useful way of moving about large programs, but beware - it can also be very dangerous because if the line really did exist before you entered the line number - it certainly wouldn't exist afterwards!

To list a program on the screen, type...

```
LIST
```

...and press **ENTER**. You may (particularly when working with more lengthy programs) wish to list from a certain point onwards. This can be achieved by typing an appropriate line number after the `LIST` command.

Type...

```
LIST 15
```

...and press **ENTER**, to see this illustrated.

When we were developing the above program, note how we were able to insert line 15 between the other two lines - this would have been impossible if they had been numbered 1 and 2 instead of 10 and 20. It is always good practice, therefore, to leave gaps between line numbers.

(Note that line numbers must be entered as whole numbers between 1 and 9999.)

If, at some time, you find that you haven't left enough space between line numbers, then you may use the edit menu to renumber a program. To do this, press the **EDIT** key then select the `Renumber` option from the menu that appears; this sets the gap between each line number to 10. Try this out and see how the line numbers change.

We are now going to use the BASIC command **NEW**. This erases any existing programs and variables in the *+3*'s memory. The command should be used whenever you are about to start afresh, so type ..

```
NEW
```

and press **ENTER**. From now on we won't mention 'press **ENTER**' every time - we'll assume that you'll remember.

With the opening menu on the screen, start up BASIC by selecting the option **+3 BASIC**.

Now carefully type in this program which converts Fahrenheit temperatures to Celsius (centigrade) ..

```
10 REM temperature conversion
20 PRINT "deg F","deg C"
30 PRINT
40 INPUT "Enter deg F",f
50 PRINT f,
60 PRINT (f-32)*5/9
70 GO TO 40
```

Although you can type in all of line 10 in lower case, only the **REM** will be converted to upper case on entry as it's the only keyword that the *+3* recognises. Also, although the words **GO TO** will appear with a space between them, they may be typed in as one word (**GOTO**) if you prefer.

Now run the program. The instructions will start being carried out in the order determined by the line numbers. First of all, you'll see the headings **deg F** and **deg C** printed on the screen (as instructed by line 20), but what has line 10 done? It looks like the *+3* has completely ignored it - in fact, it has! The **REM** in line 10 stands for remark, so line 10 is solely to remind you of what the program does. A **REM** command consists of **REM** followed by anything you like - the *+3* will ignore everything after the **REM** right up to the end of the line.

After line 20, the *+3* carries out line 30 which simply prints a blank line. When the *+3* gets to the **INPUT** command in line 40 it waits for you to type in a value for the variable **f** - you can tell this because at the bottom of the screen is a flashing cursor.

Type in a number (then press **ENTER**). The *+3* displays the result and then waits for you to enter another number. This is because the instruction in line 70 says **GO TO 40** - in other words, 'instead of running out of program and stopping, jump back to line 40 and continue running from there'.

So, enter another temperature, then another.

After a few more of these you might be wondering if the computer will ever get bored with this - it won't! Next time it asks for another number, hold down **SYMB SHIFT** and type **A**. The word **STOP** will appear, and when you press **ENTER** the *+3* comes back with the report

```
H STOP in INPUT in line 40:1
```

which tells you why it stopped, and where (in line 40). (The **:1** after the line number in the report tells you that the 1st instruction in line 40 is being reported upon.)

If you wish to continue the program, type..

```
CONTINUE
```

...and the *+3* will ask you for another number.

When **CONTINUE** is used, the *+3* remembers the line number in the last report that it sent you (as long as the report was not **0 OK**) and jumps back to that line, which in this case is line 40 (the **INPUT** command)

Stop the program again and replace line 70 by

```
70 GO TO 31
```

There will be no perceptible difference to the running of the program because if the line number in a **GO TO** command refers to a non-existent line, then the jump is to the next line after the given number. The same goes for **RUN** (in fact, **RUN** on its own actually means **RUN 0**)

Keep entering numbers until the screen starts getting full. When it is full, the *+3* will move the whole of the top half of the screen up one line to make room, losing the heading off the top - this is called ***scrolling***.

When you are tired of entering numbers, stop the program as before and enter the editor by pressing **ENTER**.

Look at the **PRINT** statement in line 50. The **,** comma in this line is very important

Commas are used to make the printing start either at the left-hand margin, or in the middle of the screen (depending upon which comes next). Thus in line 50, the comma causes the Celsius temperature to be printed in the middle of the line

A semicolon **;** on the other hand, is used to make the next number (or characters) be printed immediately after the preceding one(s)

Another punctuation mark you can use like this in **PRINT** commands is the **'** apostrophe. This makes whatever is printed next appear at the beginning of the next line on the screen. This also happens by default at the end of each **PRINT** command

If you wish to inhibit this (so that whatever follows to be printed continues on the same line) you can put a comma or semicolon at the end of the **PRINT** statement. To see how this works, replace line 50 in turn by each of these

```
50 PRINT f,
50 PRINT f;
50 PRINT f
```

.and run the program each time to see the difference.

The line with the comma (you typed in originally) prints everything in two columns; the line with the semicolon crams everything together, and the line without either, prints each number on a new line (you could have also used **PRINT f'** to do this)

Always remember the difference between commas and semicolons in **PRINT** commands, and do not confuse them with **:** colons which are used as separators between commands on a single line, for example...

```
PRINT f: GO TO 40
```

Now type in these extra lines.

```
100 REM greeting program
110 INPUT "Enter your name",n$
120 PRINT "Hello ";n$;"!"
130 GO TO 110
```

This is a separate program from the last one, but you may keep them both in the ***+3*** at the same time. To run the new one, type

```
RUN 100
```

Because this program expects you to input a ***string*** (a character or group of characters) instead of a number, it prints out two string quotes " " as a reminder. So type in a name and press **ENTER**.

Next time round, you will get two string quotes again, but you don't have to use them if you don't want to. Try this, for example: rub out the quotes by pressing cursor right ⇧ then **DELETE** twice, and type..

```
n$
```

Since there are no string quotes, the ***+3*** knows that it has to do some calculation - the calculation in this case is to find the value of the string variable called **n$** (which is whatever name you happen to have typed in last time round). In this way, the **INPUT** statement acts like **LET n$=n$**, so the value of **n$** is unchanged.

If you wish to stop the program, delete the quotes then hold down **SYMB SHIFT** and type **A**, then **ENTER**.

Now look back at that **RUN 100** instruction which jumps to line 100 and runs the program from there. You may be asking, What's the difference between **RUN 100** and **GO TO 100**? Well, **RUN 100** first of all clears all the variables and the screen, and after that works just like **GO TO 100**. On the other hand, **GO TO 100** doesn't clear anything, and there may well be occasions where you wish to run a program without clearing any variables, here **GO TO** would be necessary and **RUN** could be disastrous, so it is better not to get into the habit of automatically typing **RUN** to start a program.

Another difference of course is that you may type **RUN** without a line number, and it starts off at the first line in the program; **GO TO** must always be followed by a line number.

Both this program and the 'temperature conversion' program stopped because you pressed **SYMB SHIFT** and typed **A** in the input line. Sometimes, you may write a program that you can't stop and that won't stop itself. Type..

```
200 GO TO 200
RUN 200
```

Although the screen is blank, the program is running - executing line 200 over and over again. This looks all set to go on forever unless you pull the plug out or reset the computer! However, there is a less drastic remedy - press the **BREAK** key. The program will stop with the report

```
L BREAK into program
```

At the end of every statement, the program looks to see if this key is pressed, and if it is, then the program stops. The **BREAK** key can also be used when you are in the middle of using a printer, a cassette unit, or various other add-ons that you can attach to the ***+3***

In these cases there is a different report

```
D BREAK - CONT repeats
```

The instruction `CONTINUE` in this case (and in most other cases too) repeats the statement where the program was stopped and carries straight on with the next statement (after allowing for any jumps to be made.

Run the 'name' program again and when it asks you for input type

```
n$                    (after removing the quotes)
```

Because **n$** is an undefined variable you will get the error report

```
2 Variable not found
```

If you now type

```
LET n$="Fremsley"
```

(which produces the report `0 OK,0:1`) and then type

```
CONTINUE
```

you will find that you can use **n$** as input data without any trouble

In this case **CONTINUE** does a jump to the **INPUT** command in line 110. It disregards the report from the **LET** statement because that said **OK** and jumps to the command referred to in the previous report, ie. line 110. This feature can be extremely useful as it allows you to 'fix' a program that has stopped due to errors, and then **CONTINUE** from that point.

As we said before, the report **L BREAK into program** is special because after it, **CONTINUE** does not repeat the command where the program stopped.

You have now seen the statements **PRINT**, **LET**, **INPUT**, **RUN**, **LIST**, **GO TO**, **CONTINUE**, **NEW** and **REM**, and they can all be used either as direct commands or in program lines - this is true of almost all commands in ***+3*** BASIC however. **RUN**, **LIST**, **CONTINUE** and **NEW** are not usually of much use in a program.

Exercises:

1. Put a **LIST** statement in a program so that when you run it, it lists itself afterwards.

2. Write a program to input prices and print out the tax due (at 15 percent). Put in **PRINT** statements so that the ***+3*** announces what it is going to do, and asks for the input price with extravagant politeness. Modify the program so that you can also input the tax rate (to allow for zero ratings or future changes).

3. Write a program to print a running total of numbers you input (like an adding machine).

4. What would **CONTINUE** and **NEW** do in a program? Can you think of any uses at all for this?

Part 3
Decisions

Subjects covered...

```
CLS, IF, STOP
=, <, >, <=, >=, <>
```

All the programs we have seen so far have been pretty predictable - they went straight through the instructions, and then went back to the beginning again. This is not very useful, as in practice, we would want the *+3* to make decisions and act accordingly. The instruction to do this in BASIC takes the form...

IF something is true (or not true) **THEN** do something

Let's look at an example of this. Use **NEW** to clear the previous program from the *+3*, select **+3 BASIC**, then type in and run this program. (This is clearly meant for two people to play!)...

```
10 REM Guess the number
20 INPUT "Enter a secret numbe
   r",a: CLS
30 INPUT "Guess the number",b
40 IF b=a THEN PRINT "That is
   correct": STOP
50 IF b<a THEN PRINT "That is
   too small, try again"
60 IF b>a THEN PRINT "That is
   too big, try again"
70 GO TO 30
```

Note that the **CLS** command (at the end of line 20) means 'clear the screen'. We have used it in this program to stop the other person seeing the secret number after it is entered.

You can see that the **IF** statement takes the form...

IF condition **THEN** xxx

... where 'xxx' stands for a command (or a sequence of commands separated by colons). The condition is something that is going to be worked out as either true or false - if it comes out as true then the statements in the rest of the line (after **THEN**) are executed; otherwise they are skipped over, and the program executes the next instruction.

The simplest conditions compare two numbers or two strings, they can test whether two numbers are equal or whether one is bigger than the other. They can also test whether two strings are equal, or whether one comes before the other in alphanumerical order. They use the symbols =, <, >, <=, >=, and <> (these are known as relational operators)

=	means is equal to.
<	means is less than.
>	means is greater than.
<=	means is less than or equal to.
>=	means is greater than or equal to
<>	means is not equal to.

(If you keep getting mixed up about the meanings of < and >, it may help you to remember that the thin end of the symbol points to the number which is supposed to be smaller.)

In the program we have just typed in, line 40 compares a and b. If they are equal then the program is halted by the **STOP** command. The report at the bottom of the screen.

```
9 STOP statement, 40:3
```

shows that the 3rd statement (ie. the **STOP** command) in line 40 caused the program to halt.

Line 50 determines whether b is less than a and line [illegible] whether b is greater than a. If one of these conditions is true then the appropriate comment is printed, and the program works its way down to line 70 which jumps back to line 30 and starts all over again.

Finally, note that in some versions of BASIC (not **+3** BASIC) the **IF** statement can have the form:

IF condition **THEN** line number

This means the same as..

IF condition **THEN GO TO** line number

in **+3** BASIC

Exercise...

1. Try this program..

```
10 LET a=1
20 LET b=1
30 IF a>b THEN PRINT a;" is higher"
40 IF a<b THEN PRINT b;" is higher"
```

Before you run it, try to work out what will be printed on the screen.

Part 4
Looping

Subjects covered...

```
FOR, NEXT
TO, STEP
```

Suppose you wish to input five numbers and add them together

One way (don't type this in unless you are feeling dutiful) is as follows

```
 10 LET total=0
 20 INPUT a
 30 LET total=total+a
 40 INPUT a
 50 LET total=total+a
 60 INPUT a
 70 LET total=total+a
 80 INPUT a
 90 LET total=total+a
100 INPUT a
110 LET total=total+a
120 PRINT total
```

This method is not good programming practice. It may be just about controllable for five numbers, but you can imagine how tedious a program like this to add twenty numbers would be, and to add a hundred or more would be out of the question

Much better is to set up a variable to count up to 5 and then stop the program, like this (which you should type in)

```
 10 LET total=0
 20 LET count=1
 30 INPUT a
 40 REM count is number of time
    s that a has been input so
    far
 50 LET total=total+a
 60 LET count=count+1
 70 IF count <= 5 THEN GO TO 30
 80 PRINT total
```

Notice how easy it would be to change line 70 so that this program adds ten numbers or even a hundred

This sort of thing is so useful that there are two special commands to make it easier - the **FOR** command and the **NEXT** command. They are always used together. Using these, the program you have just typed in does exactly the same as

```
10 LET total=0
20 FOR c=1 TO 5
30 INPUT a
40 REM c is number of times th
   at a has been input so far
50 LET total=total+a
60 NEXT c
80 PRINT total
```

(To get this program from the previous one, you just have to edit lines 20, 40 and 60, then delete line 70.)

Note that we have changed **count** to **c**. This is because the control variable of a **FOR NEXT** loop must have a single letter as its name.

The effect of this program is that **c** runs through the values 1 (the initial value), 2, 3, 4 and 5 (the limit), and each time, lines 30, 40 and 50 are executed. Then, when **c** has finished its five values, line 80 is executed.

At this point, attempt exercise 2 (which refers to the above program), at the end of this section.

An extra subtlety to the **FOR NEXT** structure is that the control variable does not have to go up by 1 each time - you can change this 1 to anything you like by using a **STEP** part in the **FOR** command. The most general form of a **FOR** command is:-

FOR control variable = initial value **TO** limit **STEP** step

..where the control variable is a single letter, and where the initial value, the limit and the step are all things that the ***+3*** can calculate as numbers - like the actual numbers themselves, or sums, or the names of numeric variables. So, if you replace line 20 in the program by :

```
20 FOR c=1 TO 5 STEP 3/2
```

this will step the control variable by the amount 3/2 each time the **FOR** loop is executed. Note that we could have simply said **STEP 1.5**, or we could have assigned the step value to a variable, say **s**, and then said **STEP s**.

With the above modification, **c** will run through the values 1, 2.5 and 4. Notice that you don't have to restrict yourself to whole numbers, and also that the control value does not have to hit the limit exactly; it carries on looping as long as it is less than or equal to the limit.

At this point, attempt exercise 3 at the end of this section (which refers to the above program).

Step values can be negative instead of positive. Try this program which prints out the numbers from 1 to 10 in reverse order. (Remember use the command **NEW** before typing in a new program).

```
10 FOR n=10 TO 1 STEP -1
20 PRINT n
30 NEXT n
```

We said before that the program carries on looping as long as the control variable is less than or equal to the limit. If you consider what that would mean in this case, you'll see that it now doesn't hold true. Hence, the rule has to be modified to say that when the step is negative, the program carries on looping as long as the control variable is greater than or equal to the limit.

At this point, attempt exercises 4 and 5 at the end of this section (which refer to the above program).

You must be careful if you are running two **FOR...NEXT** loops together, one inside the other. Try this program, which prints out the numbers for a complete set of six dot dominoes.

```
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT n
50 PRINT
60 NEXT m
```

n loop

m loop

You can see that the n loop is entirely inside the m loop. This means that they are properly ***nested***.

However, what must be avoided is having two **FOR...NEXT** loops that overlap without either being entirely inside the other, like this:

```
 5 REM this program is wrong
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT m
50 PRINT
60 NEXT n
```

m loop

n loop

Two **FOR...NEXT** loops must either be one inside the other, or completely separate.

Another thing to avoid is jumping into the middle of a **FOR...NEXT** loop from the outside. The control variable is only set up properly when its **FOR** statement is executed, and if you miss this out, then the **NEXT** statement will confuse the ***+3***. You will probably get an error report saying **NEXT without FOR** or **Variable not found**.

There is nothing to stop you using a **FOR NEXT** loop in a direct command. For example, try...

```
FOR m=Ø TO 1Ø: PRINT m: NEXT m
```

You can sometimes use this as a (somewhat artificial) way of getting around the restriction that you cannot **GO TO** anywhere inside a command - because a command has no line number. For instance...

```
FOR m=Ø TO 1 STEP Ø: INPUT a: PRINT a: NEXT m
```

The step size of zero here makes the command repeat itself forever.

This sort of thing is not really recommended, because if an error crops up then you have lost the command and will have to type it in again; moreover **CONTINUE** will not work.

Exercises...

1. Make sure that you understand that a control variable not only has a name and a value (like an ordinary variable), but also a limit, a step, and a reference to the statement after the corresponding FOR statement. Ensure that when the **FOR** statement is executed all this information is available (using the initial value as the first value the variable takes), and also that this information is enough for the **NEXT** statement to know by how much to increase the value, whether to jump back, and if so where to jump back to.

2. Run the third program in this section, then type:

```
PRINT c
```

Why is the answer 6, and not 5?

(Answer: The **NEXT** command in line 60 is executed five times, each time 1 being added to c. On the last time, c becomes 6 so the **NEXT** command decides not to loop back but to carry on, c now being past its limit.)

3. What happens if you put **STEP 2** at the end of line 20 of the third program? Try **STEP 1Ø**.

Now change the third program so that instead of automatically adding five numbers, it asks you to input the amount of numbers you wish to add. When you run this program, what happens if you input **Ø** (meaning that you don't wish to add any numbers)? Why might you expect this to cause problems for the ***+3***, even though it is clear what you mean?

4. In line 10 of the fourth program in this section, change **1Ø** to **1ØØ** and run the program. It will print the numbers from 100 down to 79 on the screen, and then say **scroll?** at the bottom. This is to give you a chance to see the numbers that are about to be scrolled off the top. If you press **N**, **BREAK** or the space bar, the program will stop with the report **D BREAK - CONT repeats**. If you press any other key, then it will print another 22 lines and ask you again if you wish to scroll.

5. Delete line 30 from the fourth program. When you run the new curtailed program, it will print the first number and stop with the message `Ø OK`. If you then type

```
NEXT n
```

...the program will go once round the loop, printing out the next number.

Part 5
Subroutines

Subjects covered...

GO SUB, RETURN

Sometimes, different parts of the program will have rather similar jobs to do, and you will find yourself typing in the same lines two or more times, however this is not necessary. Instead, you need only type in the lines once (in what's called a ***subroutine***) and then call the subroutine into action whenever you need it in the program.

To do this, you use the statements **GO SUB** (go to subroutine) and **RETURN**. This takes the form:

```
GO SUB xxx
```

where xxx is the line number of the first line in the subroutine. It is just like **GO TO** xxx except that the ***+3*** remembers where the **GO SUB** statement was, so that it can come back again after carrying out the subroutine.

(In case you are interested, the ***+3*** does this by remembering at which point in the program the **GO SUB** command was issued (in other words where it should continue from afterwards) and storing this ***return address*** on top of a pile called the **GO SUB *stack***.)

When the command

```
RETURN
```

is met (at the end of the subroutine itself), the ***+3*** takes the top return address off the **GO SUB** stack, and continues from the next statement.

As an example, let's look at the number guessing program again. Retype it as follows:

```
10 REM "A rearranged guessing
   game"
20 INPUT "Enter a secret numbe
   r",a: CLS
30 INPUT "Guess the number",b
40 IF b=a THEN PRINT "Correct"
   : STOP
50 IF b<a THEN GO SUB 100
60 IF b>a THEN GO SUB 100
70 GO TO 30
100 PRINT "Try again"
110 RETURN
```

The **GO TO 30** statement in line 70 (and the **STOP** statement in line 60 of the next program) are very important because otherwise the programs will run on into their subroutines and cause an error (**7 RETURN without GO SUB**) when the **RETURN** statement is reached.

The following program uses a subroutine (from line 100 to 150) which prints a times table corresponding to the value of parameter **n**. The command **GO SUB 100** may be issued from any point in the program to call the subroutine. When the **RETURN** command in line 150 of the subroutine is reached, control returns to the main program, which continues running from the statement after the **GO SUB** call. Like **GO TO GO SUB** may be typed in as **GOSUB**.

```
 10 REM times tables for 2, 5,
    10 and 11
 20 LET n=2: GO SUB 100
 30 LET n=5: GO SUB 100
 40 LET n=10: GO SUB 100
 50 LET n=11: GO SUB 100
 60 STOP
 70 REM end of main program, st
    art of subroutine
100 PRINT n;" times table"
110 FOR t=1 TO 9
120 PRINT t;" x ";n;" = ";t*n
130 NEXT t
140 PRINT
150 RETURN
```

One subroutine can happily call another, or even itself (a subroutine that calls itself is known as *recursive*).

Part 6
Data in programs

Subjects covered..

READ, DATA, RESTORE

In some of the previous programs we saw that information, or data, can be entered directly into the *+3* using the **INPUT** statement. Sometimes this can be very tedious, especially if a lot of the data is repeated every time the program is run. You can save a lot of time by using the **READ**, **DATA** and **RESTORE** commands. For example

```
10 READ a,b,c
20 PRINT a,b,c
30 DATA 1,2,3
```

A **READ** statement consists of **READ** followed by a list of the names of variables, separated by commas. It works rather like an **INPUT** statement, except that instead of getting *you* to type in the values to give to the variables, the *+3* looks up the values in the **DATA** statement.

Each **DATA** statement is a list of expressions - numeric or string expressions - separated by commas. You can put them anywhere you like in a program, because the *+3* ignores them except when it is doing a **READ**. You must imagine the expressions from all the **DATA** statements in the program as being put together to form one long list of expressions - the **DATA** list. The first time the *+3* goes to **READ** a value, it reads the first expression from the **DATA** list, the next time, it reads the second, and thus as it meets successive **READ** statements, it works its way through the **DATA** list. (If it tries to read past the end of the **DATA** list, then it reports an error.)

Note that it's a waste of time putting **DATA** statements in a direct command, because **READ** will not find them. **DATA** statements must go in a program.

Let's see how all this works in the program you've just typed in. Line 10 tells the *+3* to read three pieces of data and assign them to the variables a, b and c. Line [illegible] then says **PRINT** these variables. The **DATA** statement in line 30 provides the values of a, b and c for line 10 to read.

The information in **DATA** can be part of a **FOR ... NEXT** loop. Type in.

```
10 DATA 2,4,6,8,10,12
20 FOR n=1 TO 6
30 READ d
40 PRINT d
50 NEXT n
```

Note from the above two programs that a **DATA** statement can appear anywhere - before or after the **READ** statement.

When the above program is run, the **READ** statement moves through the **DATA** list with each pass of the **FOR...NEXT** loop.

DATA statements may also contain string variables. For example..

```
10 FOR a=1 TO 7
20 READ n$
30 PRINT n$
40 DATA "Bob","Edith","Carole"
   ,"Jacquie","Gavin","Charles
   ","Holly"
50 NEXT a
```

The ***+3*** doesn't have to **READ** the **DATA** statements in order - it can be made to 'jump about' between **DATA** statements by using the **RESTORE** command. The form of the command is.

```
RESTORE xxx
```

...where 'xxx' is the line number of the **DATA** statement to be **READ** from. If you use the command **RESTORE** on its own (without a line number) the ***+3*** will jump to the first **DATA** statement in the program.

Type in and run the following program..

```
 10 DATA 1,2,3,4,5
 20 DATA 6,7,8,9
 30 GO SUB 110
 40 GO SUB 110
 50 GO SUB 110
 60 RESTORE 20
 70 GO SUB 110
 80 RESTORE
 90 GO SUB 110
100 STOP
110 READ a,b,c
120 PRINT a'b'c
130 PRINT
140 RETURN
```

The command **GO SUB 110** calls a subroutine which **READ**s the next three items of **DATA** and then **PRINT**s them. Notice how the **RESTORE** command affects which items are read.

Delete line 60 and run this program again to see what happens

Part 7
Expressions

Subjects covered...

Operations: +, -, *, /
Expressions, scientific notation, variable names

You have already seen some of the ways in which the *+3* can calculate with numbers. It can perform the four arithmetic operations + - * and / (remember that * is used for multiplication, and / is used for division), and it can find the value of a variable, given its name.

The example

```
LET tax=sum*15/100
```

...illustrates that calculations can be combined. Such a combination, like `sum*15/100`, is called an expression - so an ***expression*** is just a short-hand way of telling the *+3* to do several calculations, one after the other. In our example, the expression `sum*15/100` means 'look up the value of the variable called `sum`, multiply it by 15, and divide by 100'.

In expressions containing * / + - multiplication and division are carried out first - they have a higher priority than addition and subtraction. Multiplication and division have the same priority as each other, which means that they are carried out in whichever order they appear in the expression (from left to right). The next operations to be carried out are addition and subtraction - these again have the same priority as each other and so, again, are carried out in order from left to right.

Hence in the expression `8-12/4+2*2` the first operation to be carried out is the division `12/4` which equals 3, so we can then represent the expression as `8-3+2*2`.

The next operation to be carried out is the multiplication `2*2` which equals 4, so the expression then becomes `8-3+4`.

Next to be carried out is the subtraction `8-3` which equals 5, so the expression becomes `5+4`. Finally, the addition is carried out leaving the result 9.

Try this out for yourself. Type in:

```
PRINT 8-12/4+2*2
```

A full list of the priorities of mathematical (and logical) operations will be found in part 31 of this chapter.

You may, however, change the priority of calculations within an expression by the use of brackets. Calculations within brackets are carried out first, so if in the above expression, you required the addition 4+2 to be carried out first, you would enclose it in brackets. To see this, type in...

```
PRINT 8-12/(4+2)*2
```

...and the result this time is 4 instead of 9.

Expressions are useful because, whenever the ***+3*** is expecting a number from you, you can give it an expression instead and it will work out the answer.

You can also add together strings (or string variables) in a single expression. For example...

```
10 LET a$="large "
20 LET b$="and puffy"
30 LET c$=a$+b$
40 PRINT c$
```

We really ought to tell you what you can and cannot use as the names of variables. As we have already said, the name of a string variable has to be a single letter followed by **$**, and the name of the control variable in a **FOR NEXT** loop must be a single letter, however, the names of ordinary numeric variables are less restricted - they can use any letters or digits as long as the first one is a letter. You can put spaces in as well to make it easier to read, but they won't count as part of the name. Also, it doesn't make any difference to the name whether you type it in upper or lower case letters. There are some restrictions about variable names which are the same as commands, however. In general, if the variable contains a BASIC keyword in it (with spaces around it) then it won't be accepted.

Here are some examples of the names of variables that ***are*** allowed.

```
x
any old thing
t42
this name is impractical because it is too long
tobeornottobe
mixed cases spaces
MixEdCAsEsSpAcES
```

(Note that these last two names (**mixed cases spaces** and **MixEdCAsEsSpAcES**) are considered the same, and refer to the same variable)

The following are *not* allowed as the names of variables..

`pi`	(PI is a keyword)
`any new thing`	(contains the separated keyword `NEW`)
`42t`	(begins with a digit)
`2001`	(contains digits only)
`to be or not to be`	(contains `TO` `OR` and `NOT` which are all separated keywords)
`3 bears`	(begins with a digit)
`M*A*S*H`	(* is not a letter or a digit)
`Lloyd-Webber`	(- is not a letter or a digit)

Numerical expressions can be represented by a number and exponent. Try the following to prove the point..

```
PRINT 2.34e0
PRINT 2.34e1
PRINT 2.34e2
```

..and so on up to

```
PRINT 2.34e15
```

`PRINT` gives only eight significant digits of a number. Try

```
PRINT 4294967295, 4294967295-429e7
```

This proves that the computer can hold the digits of 4294967295, even though it is not prepared to display them all at once.

The *+3* uses ***floating point arithmetic***, which means that it keeps separate the digits of a number (its ***mantissa***) and the position of the point (the ***exponent***). This is not always exact, even for whole numbers. Type..

```
PRINT 1e10+1-1e10,1e10-1e10+1
```

Numbers are held to about nine and a half digits accuracy, so 1e10 is too big to be held exactly right. The inaccuracy (actually about 2) is more than 1, so the numbers 1e10 and 1e10+1 appear to the computer to be equal.

For an even more peculiar example, type

```
PRINT 5e9+1-5e9
```

Here the inaccuracy in 5e9 is only about 1, and the 1 to be added on in fact gets rounded up to 2. The numbers 5e9+1 and 5e9+2 appear to the computer to be equal. The largest ***integer*** (whole number) that can be held completely accurately is 4,294,967,294.

The string "" with no character at all is called the ***empty*** or ***null string***. Remember that spaces are significant and an empty string is not the same as one containing nothing but spaces.

Try

```
PRINT "Did you read "The Times" yesterday?"
```

When you press **ENTER** you will get the flashing red cursor that shows there is a mistake somewhere in the line. When the ***+3*** finds the double quotes at the beginning of `"The Times"` it imagines that these mark the end of the string `"Did you read "` and it then can't work out what `The Times` means.

There is a special device to get over this: whenever you wish to write a string quote symbol in the middle of a string, you must write it twice, like this:

```
PRINT "Did you read ""The Times"" yesterday?"
```

As you can see from what is printed on the screen, each double quote is only really there once - you just have to type it twice to get it recognised.

Part 8
Strings

Subjects covered..

Slicing, using TO

Given a string, a *substring* of it consists of some consecutive characters in it, taken in sequence. Thus `"cut"` is a substring of `"cutlery"`, but `"cute"` and `"cruelty"` are not substrings.

There is a notation called *slicing* for dealing with substrings, and it can be applied to arbitrary string expressions. The general form is

string expression(start TO finish)

so that, for instance,

```
"abcdef"(2 TO 5)
```

is equal to `bcde`.

If you omit the start, then it is assumed to be the first, and if you omit the finish, then the length of the string is assumed. Thus

```
"abcdef"( TO 5)    is equal to abcde
"abcdef"(2 TO )    is equal to bcdef
"abcdef"( TO )     is equal to abcdef
```

You can also write the last one as `"abcdef"()`.

A slightly different form omits out the `TO` and just has one number:

```
"abcdef"(3) is equal to "abcdef"(3 TO 3) is equal to c
```

Although normally the [illegible] overridden by another one if the [illegible]

```
"abcdef"(5 TO 7)
```

gives the error `3 Subscript wrong` because the string has only six characters and 7 is too many, but

```
"abcdef"(8 TO 7)
```

...and...

```
"abcdef"(1 TO 0)
```

.. are both equal to the empty string "" and are therefore permitted.

The start and finish must not be negative, or you get the error **B integer out of range**. This next program is a simple one illustrating some of these rules.

```
10 LET a$="abcdef"
20 FOR n=1 TO 6
30 PRINT a$(n TO 6)
40 NEXT n
```

Type **NEW** when this program has been run and enter the next program

```
10 LET a$="1234567890"
20 FOR n=1 TO 10
30 PRINT a$(n TO 10),a$((11-n)
    TO 10)
40 NEXT n
```

For string variables we can not only extract substrings, but also assign to them. For instance, type

```
LET a$="Velvet Donkey"
```

..and then,

```
LET a$(8 TO 13)="Lips******"
```

and

```
PRINT a$
```

Since the substring **a$(8 TO 13)** is only 6 characters long only its first 6 characters **(Lips**)** are used, the remaining 4 characters (****) are discarded. This is a characteristic of assigning to substrings: the substring has to be exactly the same length afterwards as it was before. To make sure this happens, the string that is being assigned to it is cut off on the right if it is too long, or filled out with spaces if it is too short - this is called 'Procrustean assignment' after the inn-keeper Procrustes who used to make sure that his guests fitted their beds by either stretching them out on a rack or cutting their feet off!

Complicated string expressions will need brackets around them before they can be sliced. For example...

```
"abc"+"def"(1 TO 2)        is equal to "abcde"
("abc"+"def")(1 TO 2)      is equal to "ab"
```

Exercise...

1. Try writing a program to print the day of the week using string slicing. (Hint - Let the string be `SunMonTuesWednesThursFriSatur`).

Part 9
Functions

Subjects covered...

```
LEN, STR$, VAL, SGN, ABS, INT, SQR
DEF FN
```

Consider the sausage machine. You put a lump of meat in at one end, turn a handle and out comes a sausage at the other end. A lump of pork gives a pork sausage, a lump of fish gives a fish sausage, and a lump of beef a beef sausage.

Functions are practically indistinguishable from sausage machines but there is a difference; they work on numbers and strings instead of meat. You supply one value (called the ***argument***), mince it up by doing some calculations on it, and eventually get another value - the ***result***.

Meat in	→	Sausage Machine	→	Sausage out
Argument in	→	Function	→	Result out

Different arguments give different results, and if the argument is completely inappropriate the function will stop and give an error report.

Just as you can have different machines to make different products - one for sausages, another for combs, a third for dish cloths, and so on, different functions will do different calculations. Each will have its own value to distinguish it from the others.

You use a function in expressions by typing its name followed by the argument, and when the expression is evaluated the result of the function will be worked out.

As an example, there is a function called `LEN` which works out the length of a string. Its argument is the string whose length you wish to find, and its result is the length, so that if you type

```
PRINT LEN "Jammy Smears"
```

the ***+3*** will write the answer 12, ie the number of characters (including spaces) in the string `Jammy Smears`.

If you mix functions and operations in a single expression, then the functions will be worked out before the operations. Again, however, you can circumvent this rule by using brackets. For instance, here are two expressions which differ only in the brackets, and yet calculations are performed in an entirely different order in each case (although, as it happens, the end results are the same).

```
LEN "Fred" + LEN "Bloggs"

4+LEN "Bloggs"

4+6

10
```

and

```
LEN ("Fred" + "Bloggs")

LEN ("FredBloggs")

LEN "FredBloggs"

10
```

Here are some more functions.

STR$ converts numbers into strings. Its argument is a number, and its result is the string that would appear on the screen if the number were displayed by a **PRINT** statement. Note how its name ends in a **$** sign to show that its result is a string. For example, you could say

```
LET a$= STR$ 1e2
```

which would have exactly the same effect as typing

```
LET a$="100"
```

Or you could say

```
PRINT LEN STR$ 100.0000
```

and get the answer 3 because **STR$ 100.0000** is equal to 100, the length of which is 3 characters.

VAL is like **STR$** in reverse - it converts strings into numbers. For instance,

```
VAL "3.5"
```

is equal to the number 3.5

VAL is the reverse of **STR$** because if you take any number, apply **STR$** to it, and then apply **VAL** to it, you get back to the number you first thought of.

However, if you take a string, apply **VAL** to it, and then apply **STR$** to it, you do not always get back to your original string.

VAL is an extremely powerful function, because the string which is its argument is not restricted to looking like a plain number - it can be any numeric expression. Thus, for instance,

```
VAL "2*3"
```

...is equal to 6. Even...

```
VAL ("2"+"*3")
```

...is equal to 6. There are two processes at work here. In the first, the argument of **VAL** is evaluated as a string - the string expression **"2"+"*3"** is evaluated to give the string **"2*3"**. Then, the string has its double quotes stripped off, and what is left is evaluated as a number: so **2*3** is evaluated to give the number 6.

There is another function, rather similar to **VAL**, though probably less useful, called **VAL$**. Its argument is still a string, but its result is also a string. To see how this works, recall how **VAL** goes in two steps: first its argument is evaluated as a string, then the string quotes stripped off this, and whatever is left is evaluated as a number. With **VAL$** the first step is the same, but after the string quotes have been stripped off in the second step whatever is left is evaluated as another string. Thus...

VAL$ """Ursula""" is equal to **Ursula**

(Notice how the string quotes proliferate again.) Try:

```
LET a$="99"
```

...and print all of the following: **VAL a$**, **VAL "a$"**, **VAL """a$"""**, **VAL$ a$**, **VAL$ "a$"** and **VAL$ """a$"""**. Some of these will work, and some of them won't - try to explain all the answers.

SGN is the sign function (sometimes called ***signum***). It is the first function you have seen that has nothing to do with strings, because both its argument and its result are numbers. The result is +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

ABS is another function whose argument and result are both numbers. It converts the argument into a positive number (which is the result) by forgetting the sign, so that for instance

```
ABS -3.2
```

...is equal to

```
ABS 3.2
```

... which is simply equal to 3.2.

INT stands for ***integer*** part - an integer is a whole number, possibly negative. This function converts a fractional number into an integer by 'throwing away' the fractional part, so that for instance.

```
INT 3.9
```

...is equal to 3.

Be careful when you are applying it to negative numbers, because it always rounds down. Thus for instance

```
INT -3.1
```

..is equal to -4.

SQR calculates the square root of a number, ie. the result that, when multiplied by itself, gives the argument, for instance

```
SQR 4
```

..is equal to 2 because 2 x 2 is equal to 4

```
SQR 0.25
```

...is equal to 0.5 because 0.5 x 0.5 is equal to 0.25.

```
SQR 2
```

is equal to 1.4142136 (approx) because 1.4142136 x 1.4142136 is equal to 2 (almost).

If you multiply any number (even a negative one) by itself, the answer is always positive. This means that negative numbers do not have square roots, so if you apply **SQR** to a negative argument you get the error report `A Invalid Argument`

You can also define functions of your own. Possible names for these are **FN** followed by a letter (if the result is a number) or **FN** followed by a letter followed by **$** (if the result is a string). These functions are much stricter about brackets - the argument *must* be enclosed in brackets.

You define a function by putting a **DEF** statement somewhere in the program. For instance, here is the definition of a function **FN s** whose result is the square of the argument :

```
10 DEF FN s(x)=x*x: REM the sq
   uare of x
```

The **s** following the **DEF FN** is the name of the function. The **x** in brackets is a name by which you wish to refer to the argument of the function. You can use any single letter you like for this (or, if the argument is a string, a single letter followed by **$**)

After the = sign comes the actual definition of the function. This can be any expression, and it can also refer to the argument using the name you've given it (in this case, **x**) as though it were an ordinary variable.

When you have entered this line, you can invoke the function just like one of the *+3*'s own functions, by typing its name, `FN s`, followed by the argument. Remember that when you have defined a function yourself, the argument must be enclosed in brackets. Try it out a few times.

```
PRINT FN s(2)
PRINT FN s(3+4)
PRINT 1+ INT FN s ( LEN "chicken"/2+3)
```

Once you have put the corresponding `DEF` statement into the program, you can use your own functions in expressions just as freely as you can use the computer's.

`INT` always rounds down. To round to the ***nearest*** integer, add 0.5 first - you could write your own function to do this:

```
20 DEF FN r(x)= INT (x+0.5): R
   EM gives x rounded to the n
   earest integer.
```

You will then get, for instance:

```
FN r(2.9)      is equal to 3
FN r(2.4)      is equal to 2
FN r(-2.9)     is equal to -3
FN r(-2.4)     is equal to -2
```

Compare these with the answers you will get when you use `INT` instead of `FN r`. Type in and run the following:

```
10 LET x=0: LET y=0: LET a=10
20 DEF FN p(x,y)=a+x*y
30 DEF FN q()=a+x*y
40 PRINT FN p(2,3), FN q()
```

There are a lot of subtle points in this program. Firstly, a function is not restricted to just one argument; it can have more, or even none at all - but you must still always keep the brackets.

Secondly, it doesn't matter whereabouts in the program you put the `DEF` statements. After the *+3* has executed line 10, it simply skips over lines 20 and 30 to get to line 40. They do, however, have to be somewhere in the program - they can't be in a command.

Thirdly, `x` and `y` are both the names of variables in the program as a whole, and the names of arguments for the function `FN p`. `FN p` temporarily forgets about the variables called `x` and `y`, but since it has no argument called `a`, it still remembers the variable `a`. Thus when `FN p(2,3)` is being evaluated, `a` has the value 10 because it is the variable, `x` has the value 2 because it is the first argument, and `y` has the value 3 because it is the second argument. The result is then `10+2*3` which is equal to 16. When `FN q()` is being evaluated, on the other hand, there are no arguments, so `a`, `x` and `y` all still refer to the variables and so have the values 10 and 0 respectively. The answer in this case is `10+0*0` which is equal to 10.

Now change line 20 to

```
20 DEF FN p(x,y)= FN q()
```

This time, FN p(2,3) will have the value 10 because FN q will still go back to the variables x and y rather than using the arguments of FN p

Some BASICs (not +3 BASIC) have functions called LEFT$, RIGHT$ MID$ and TL$

LEFT$(a$,n) gives the substring of a$ consisting of the first n characters

RIGHT$(a$,n) gives the substring of a$ consisting of the characters from nth on.

MID$(a$,n1,n2) gives the substring of a$ consisting of n2 characters starting at the n1th

TL$(a$) gives the substring of a$ consisting of all its characters except the first.

You can write some user-defined functions to do the same.

```
10 DEF FN t$(a$)=a$(2 TO ): RE
   M TL$
20 DEF FN l$(a$,n)=a$( TO n):
   REM LEFT$
```

Check that these work with strings of length 0 or 1. Note that our FN l$ has two arguments, one a number and the other a string. A function can have up to 26 numeric arguments (why 26?) and at the same time up to 26 string arguments.

Exercise

Use the function FN s(x)=x*x to test SQR. You should find that.

```
FN s( SQR x)
```

equals x if you substitute any positive number for x, and

```
SQR FN s(x)
```

equals ABS x whether x is positive or negative (Why is the ABS there?)

Part 10
Mathematical functions

Subjects covered...

↑
PI, EXP, LN, SIN, COS, TAN, ASN, ACS, ATN

This section deals with the mathematics that the +3 can handle. Quite possibly you will never have to use any of this at all, so if you find it too heavy going, don't be afraid of skipping it. It covers the operation ↑ (raising to a power), the functions EXP and LN and the trigonometrical functions SIN COS, TAN and their inverses ASN ACS, and ATN

↑ and EXP

You can raise one number to the power of another. This means 'multiply the first number by itself the second number of times'. This is normally shown by writing the second number just above and to the right of the first number; but obviously this would be difficult on a computer so we use the symbol ↑ instead. For example the powers of 2 are

2 ↑ 1 equals 2
2 ↑ 2 equals 2 x 2 equals 4 (2 squared, normally written 2^2)
2 ↑ 3 equals 2 x 2 x 2 equals 8 (2 cubed normally written 2^3)
2 ↑ 4 equals 2 x 2 x 2 x 2 equals 16 (2 to the power of four, normally written 2^4)
..and so on.

Thus, at its most elementary level, a ↑ b means 'a multiplied by itself b times', but obviously this only makes sense if b is a positive whole number. To find a definition that works for other values of b, we consider the rule

a ↑ (b+c) equals a ↑ b x a ↑ c

(Notice that we give ↑ a higher priority than multiplication and division so that when there are several operations in one expression, ↑ is evaluated before * and /). You should not need much convincing that this works when b and c are both positive whole numbers, but if we decide that we want it to work even when they are not then we find ourselves compelled to accept that.

a ↑ 0 equals 1
a ↑ (-b) equals 1/a ↑ b
a ↑ (1/b) equals the bth root of a, which is to say, the number that you have to multiply by itself b times to get a

..and

a ↑ (b x c) equals (a ↑ b) ↑ c

If you have never seen any of this before then don't try to remember it straight away, just remember that...

a ↑ (-1) equals 1/a

...and...

a ↑ (1/2) equals **SQR** a

and maybe when you are familiar with these, the rest will begin to make sense.

Experiment with all this by trying this program

```
10 INPUT a,b,c
20 PRINT a*(b+c),a↑b*a↑c
30 GO TO 10
```

Of course, if the rule we gave earlier is true, then each time round, the two numbers that the ***+3*** prints out will be equal. (Note - because of the way the computer works out ↑, the number on the left, **a** in this case, must never be negative.)

A rather typical example of what this function can be used for is that of compound interest. Suppose you keep some of your money in a building society and they give 15% interest per year. Then after one year you will have not just the 100% that you had anyway, but also the 15% interest that the building society has given you, making altogether 115% of what you had originally. To put it another way, you have multiplied your sum of money by 1.15, and this is true however much you had there in the first place. After another year, the same will have happened again, so that you will then have 1.15 x 1.15, or in other words, 1.15 ↑ 2, or in other words, 1.3225 times your original sum of money. In general then, after y years, you will have 1.15 ↑ y times what you started out with.

If you try this command :

```
FOR y=0 TO 100: PRINT y,10*1.15↑y: NEXT y
```

...you will see that even starting off from just £10, it all mounts up quite quickly, and what's more, it gets faster and faster as time goes on (though you might still find that it doesn't keep up with inflation).

This sort of behaviour, where after a fixed interval of time some quantity multiplies itself by a fixed proportion, is called ***exponential growth***, and it is calculated by raising a fixed number to the power of the time.

Suppose you did this..

```
10 DEF FN a(x)=a↑x
```

Here, **a** is more or less fixed, by **LET** statements - its value will correspond to the interest rate, which changes only every so often.

There is a certain value for a that makes the function FN a look especially pretty to the trained eye of a mathematician, and this value is called *e*. The *+3* has a function called EXP defined by...

```
EXP x is equal to e ↑ x
```

Unfortunately, *e* itself is not an especially pretty number. It is an infinite non-recurring decimal. You can see its first few decimal places by typing.

```
PRINT EXP 1
```

because EXP 1 is equal to *e* ↑ 1 which is equal to *e*. Of course, this is just an approximation. You can never write down *e* exactly.

LN

The inverse of an exponential function is a logarithmic function: the logarithm (to base a) of a number x is the power to which you'd have to raise a to get the number x, and this is written $\log_a x$. Thus by definition, a ↑ $\log_a x$ is equal to x, and it is also true that log (a ↑ x) is equal to x.

You may well already know how to use base 10 logarithms for doing multiplications; these are called common logarithms. The *+3* has a function LN which calculates logarithms to the base *e*; these are called natural logarithms. To calculate logarithms to any other base, you must divide the natural logarithm by the natural logarithm of the base: ie $\log_a x$ is equal to LN x / LN a.

PI

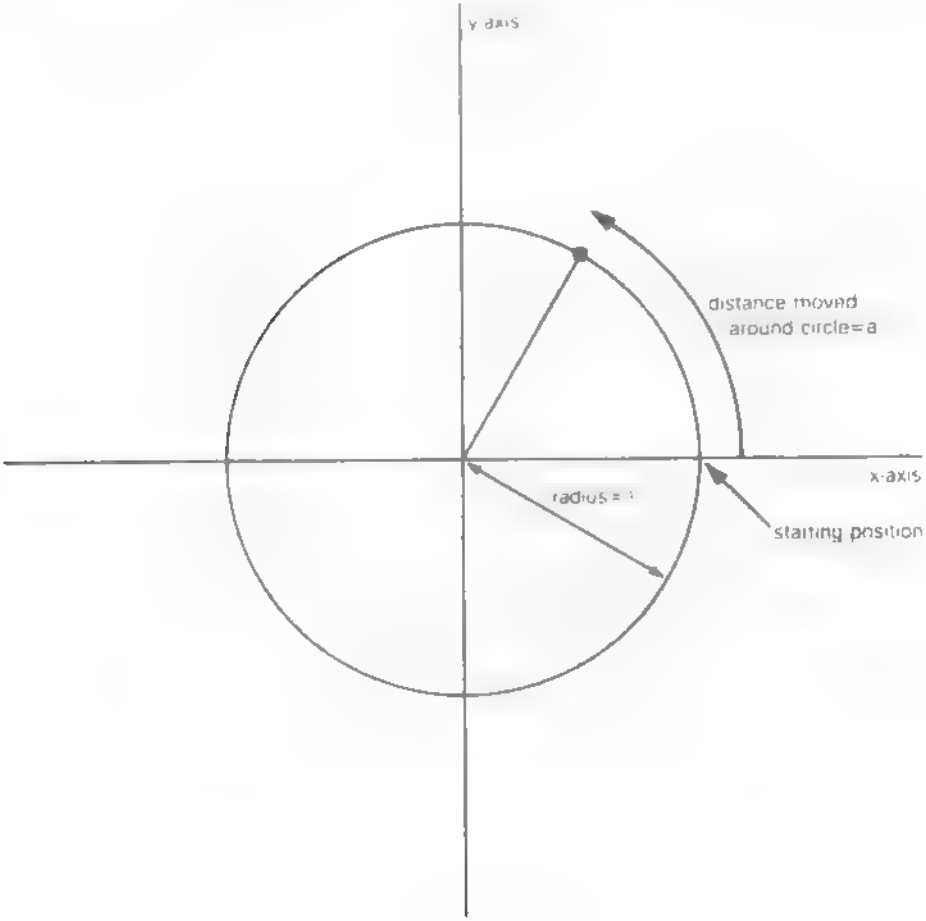
Given any circle, you can find its ***perimeter*** (the distance round its edge - often called its ***circumference***) by multiplying its diameter (width) by a number called π. π (pronounced pi) is the Greek equivalent of the English letter p, and it is used because it stands for perimeter.

Like *e*, π is an infinite non-recurring decimal - it starts off as 3.1415927. The word PI on the *+3* is taken as standing for this number. Try

```
PRINT PI
```

SIN COS and TAN, ASN ACS and ATN

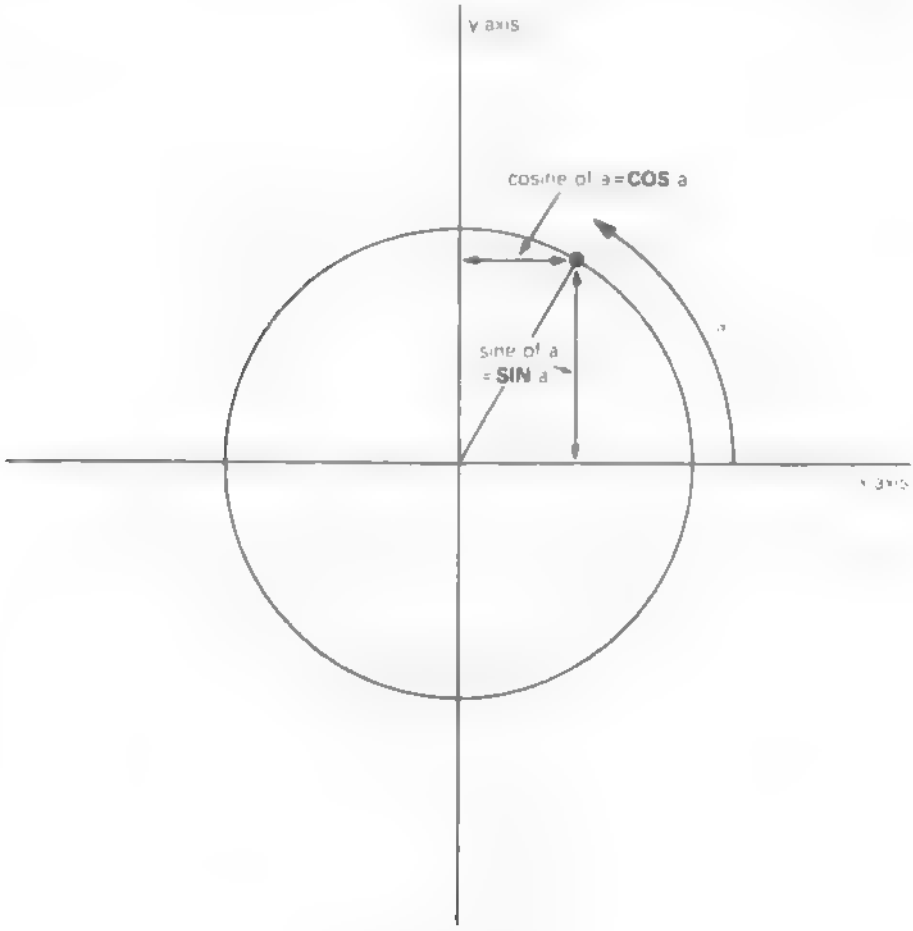
These ***trigonometrical*** functions measure what happens when a point moves round a circle. Here is a circle of radius 1 ('1 what?' you may ask - it doesn't matter, as long as we keep to the same unit all the way through) and a point moving round it. The point started at the 3 o'clock position and then moved round in an anti-clockwise direction.



We have also drawn in two lines called **axes** through the centre of the circle. The one through 3 o'clock is called the ***x-axis***, and the one through 12 o'clock is called the ***y-axis***.

To specify where the point is, you say how far it has moved round the circle from its 3 o'clock starting position. let us call this distance a. We know that the circumference of the circle is 2π (because its radius is 1 and its diameter is thus 2); so when it has moved a quarter of the way round the circle, a is equal to $\pi/2$; when it has moved halfway round, a is equal to π; and when it has moved the whole way round, a is equal to 2π.

Given the curved distance round the edge - a, two other distances you might like to know are how far the point is to the right of the y-axis, and how far it is above the x-axis. These are called, respectively, the ***cosine*** and ***sine*** of a. The functions **COS** and **SIN** on the ***+3*** will calculate these.



Note that if the point goes to the left of the y-axis, then the cosine becomes negative, and if the point goes below the x-axis, the sine becomes negative

Another property is that once a has got up to 2π, the point is back where it started and the sine and cosine start taking the same values all over again, ie `SIN (a+2*PI)` equals `SIN a`, and `COS (a+2*PI)` equals `COS a`.

The *tangent* of a is defined as being the sine divided by the cosine, the corresponding function on the ***+3*** is called `TAN`

Sometimes we need to work these functions out in reverse: finding the value of a that has given sine, cosine or tangent. The functions to do this are called ***arcsine*** (`ASN` on the ***+3***), ***arccosine*** (`ACS`) and ***arctangent*** (`ATN`)

In the diagram of the point moving round the circle, look at the radius joining the centre to the point. You should be able to see that the distance we have called a (the distance that the point has moved round the edge of the circle) is a way of measuring the angle through which the radius has moved away from the x-axis. When a is equal to $\pi/2$, the angle is 90 degrees; when a is equal to π the angle is 180 degrees, and so on, round to when a is equal to 2π, and the angle is 360 degrees. You might just as well forget about degrees, and measure the angle in terms of a alone, we say then that we are measuring the angle in ***radians***. Thus $\pi/2$ radians is equal to 90 degrees and so on.

You must always remember that on the ***+3***, the functions `SIN` `COS`, etc use radians and not degrees. To convert degrees to radians, divide by 180 and multiply by π, to convert back from radians to degrees, you divide by π and multiply by 180

Part 11
Random Numbers

Subjects covered...

```
RANDOMIZE
RND
```

This section deals with the keywords **RND** and **RANDOMIZE**

In some ways **RND** is like a function - it does calculations and produces a result. It is unusual in that it does not need an argument.

Each time you use it, its result is a new random number between 0 and 1. (Sometimes it can take the value 0, but never 1.)

Try...

```
10 PRINT RND
20 GO TO 10
```

to see how the answer varies. Can you detect any pattern? You shouldn't be able to - 'random' means that there is no pattern.

Actually, **RND** is not truly random, because it follows a fixed sequence of 65536 numbers. However, these are so thoroughly jumbled up that there are at least no obvious patterns, and we say that **RND** is *pseudo-random*.

RND gives a random number between 0 and 1, but you can easily get random numbers in other ranges. For instance, **5*RND** is between 0 and 5, and **1.3+0.7*RND** is between 1.3 and 2. To get whole numbers use **INT** (remembering that **INT** always rounds down) as in **1+INT(RND*6)** which we shall use in a program to simulate dice. **RND*6** is in the range 0 to 6 but since it never actually reaches 6, **INT(RND*6)** is 0, 1,2,3,4 or 5.

Here is the program:

```
10 REM dice throwing program
20 CLS
30 FOR n=1 TO 2
40 PRINT 1+ INT ( RND *6);" ";
50 NEXT n
60 INPUT a$: GO TO 20
```

Press **ENTER** each time you wish to throw the dice.

The **RANDOMIZE** statement may be used to make **RND** start off at a definite place in its sequence of numbers, as you can see with this program.

```
10 RANDOMIZE 1
20 FOR n=1 TO 5: PRINT RND ,:
   NEXT n
30 PRINT : GO TO 10
```

After each execution of **RANDOMIZE 1**, the **RND** sequence starts off again with 0.0022735596. You can use other numbers between 1 and 65535 in the **RANDOMIZE** statement to start the **RND** sequence off at different places.

If you had a program with **RND** in it and it also had some mistakes that you had not found, then it would help to use **RANDOMIZE** like this so that the program behaved the same way each time you ran it.

RANDOMIZE used on its own (or **RANDOMIZE 0**) have a different effect - they really do randomise **RND** - you can see this in the next program.

```
10 RANDOMIZE
20 PRINT RND : GO TO 10
```

The sequence you get here is not very random, because **RANDOMIZE** uses the time since the ***+3*** was switched on. As this has gone up by the same amount each time that **RANDOMIZE** is executed, the next **RND** does more or less the same. You would get better 'randomness' by replacing **GO TO 10** by **GO TO 20**.

Here is a program to toss coins and count the numbers of heads and tails.

```
10 LET heads=0: LET tails=0
20 LET coin= INT ( RND *2)
30 IF coin=0 THEN LET heads=he
   ads+1
40 IF coin=1 THEN LET tails=ta
   ils+1
50 PRINT heads;",";tails,
60 IF tails <> 0 THEN PRINT he
   ads/tails;
70 PRINT: GO TO 20
```

The ratio of heads to tails should become approximately 1 if you go on long enough, because in the long run you expect approximately equal numbers of heads and tails.

Exercise

1. Choose a number between 1 and 872 and type

RANDOMIZE your number

Note that the next value of **RND** will be :

```
(75*(your number+1)-1)/65536
```

Try this out for yourself.

Part 12
Arrays

Subjects covered...

Arrays
DIM

Suppose that you have a list of numbers - for instance the marks of ten people in a class. To store them in the *+3* you could use the variables m1 m2, m3 and so on up to m10 but the program to set up these ten variables would be rather long and tedious to type in, ie

```
 10 LET m1=75
 20 LET m2=44
 30 LET m3=90
 40 LET m4=38
 50 LET m5=55
 60 LET m6=64
 70 LET m7=70
 80 LET m8=12
 90 LET m9=75
100 LET m10=60
```

Instead, there is a mechanism, known as an ***array*** whereby you may specify a variable which (instead of containing a single value as variables normally do) may contain a number of separate ***elements***, each of which may contain different values. Each element is referenced by an ***index*** number (the ***subscript***) written in brackets after the variable name. For the above example the array variable's name could be m - (the name of an array variable must be a single letter), and the ten variables would then be m(1), m(2) m(3) and so on up to m(10)

The elements of an array are called ***subscripted variables*** as opposed to the simple variables that you are already familiar with

Before you can use an array you must reserve some space for it in the *+3*'s memory, and you do this by using the keyword DIM (for dimension). The statement

```
DIM m(10)
```

...sets up an array called m whose dimensions are 10 (ie there are 10 subscripted variables). The DIM statement initialises each element in the array to zero. It also deletes any array called m that existed previously - (however it doesn't delete any simple variable called m. An array variable can coexist alongside a simple numerical variable of the same name because the array can always be distinguished by its subscript)

The array elements' subscripts may be represented by any numerical expression yielding a valid subscript number. This means that an array can be processed using a FOR...NEXT loop. Thus, instead of the above long-winded program, we can now set up the variables m(1)...m(10) using:

```
10 DIM m(10)
20 FOR n=1 TO 10
30 READ m(n)
40 NEXT n
50 DATA 75,44,90,38,55,64,70,1
   2,75,60
```

...to READ in the elements' values from a DATA list, or

```
10 DIM m(10)
20 FOR n=1 TO 10
30 INPUT m(n)
40 NEXT n
```

...to INPUT the elements' values by hand.

Note that the DIM statement must come before any attempt to access the array in a program.

If you wish, you may examine the contents of the array using:

```
10 FOR n=1 TO 10
20 PRINT m(n)
30 NEXT n
```

...or individually using...

```
PRINT m(1)
PRINT m(2)
PRINT m(3)
```

...etc.

You can also set up arrays with more than one dimension. In a two dimensional array you need two numbers to specify an element - rather like the line and column numbers that specify a character position on the screen. If you imagine the line and column numbers (two dimensions) as referring to a printed page, you could then, for example, have an extra dimension to represent the page numbers. Think of the elements of a three dimensional array v as being specified by v(page number,line number,column number).

For example, to set up a two-dimensional array **c** with dimensions 3 and 6, you use the **DIM** statement...

```
DIM c(3,6)
```

This then gives you 3x6 = 18 subscripted variables...

```
c(1,1), c(1,2) ... to c(1,6)
c(2,1), c(2,2) ... to c(2,6)
c(3,1) c(3,2) ... to c(3,6)
```

The same principle works for any number of dimensions.

Although you can have a number and an array with the same name, you cannot have two arrays with the same name, even if they have a different number of dimensions.

There are also ***string arrays***. The strings in an array differ from simple strings in that they are of ***fixed length*** and assignment to them is always Procrustean (ie. chopped off or padded with spaces).

The name of a string array is a single letter followed by **$**. Unlike numeric arrays, a string array and a simple string variable ***cannot*** have the same name.

Suppose then, that you want an array **a$** of five strings. You must decide how long these strings are to be - let us suppose that 10 characters for each element is long enough. You then say...

```
DIM a$(5,10)      (type this in)
```

This sets up a 5x10 array of characters, but you can also think of each row as being a string...

```
a$(1) equals a$(1,1) a$(1,2) ... to a$(1,10)
a$(2) equals a$(2,1) a$(2,2) ... to a$(2,10)
a$(3) equals a$(3,1) a$(3,2) ... to a$(3,10)
a$(4) equals a$(4,1) a$(4,2) ... to a$(4,10)
a$(5) equals a$(5,1) a$(5,2) ... to a$(5,10)
```

If you give the same number of subscripts (two in this case) as there were dimensions in the **DIM** statement, then you get a single character, but if you miss the last one out, then you get a fixed length string. So, for instance, **a$(2,7)** is the 7th character in the string **a$(2)**. Using the slicing notation, we could also write this as **a$(2)(7)**. Now type...

```
LET a$(2)="1234567890"
```

...and...

```
PRINT a$(2), a$(2,7)
```

You get...

```
1234567890      7
```

For the last subscript (the one you can miss out), you can also have a slicer, so that for instance

`a$(2,4 TO 8)` is equal to `a$(2)(4 TO 8)` is equal to `"45678"`

Remember - In a string array, all the strings have the same fixed length.

The `DIM` statement has an extra number (the last one) to specify this length. When you write down a subscripted variable for a string array, you can put in an extra number (a slicer) to correspond with the extra number in the `DIM` statement.

You can have string arrays with no extra dimensions. Type.

```
DIM a$(10)
```

and you will find that `a$` behaves just like a string variable, except that it always has length 10, and assignment to it is always Procrustean.

Exercise.

1. Use `READ` and `DATA` statements to set up an array `m$` of twelve strings in which `m$(n)` is the name of the nth month. (Hint - The `DIM` statement will be `DIM m$(12,9)`. Test it by printing out all the values of `m$(n)` (use a loop).)

Part 13
Conditions

Subjects covered...

```
AND, OR
NOT
```

We saw in part 3 of this chapter how an **IF** statement takes the form...

```
IF condition THEN...
```

The conditions there were the relations =, <, >, <= >= and <> which compare two numbers or two strings. You can also combine several of these, using the logical operations: **AND OR** and **NOT**

One relation **AND** another relation is true whenever ***both*** relations are true, so you could have a line like

```
IF a$="yes" AND x>0 THEN PRINT "result"
```

...in which **result** gets printed only if **a$** is equal to **yes** and **x** is greater than zero. The BASIC here is so close to English that it hardly seems worth spelling out the details. As in English, you can join lots of relations together with **AND**, and then the whole lot is true if ***all*** the individual relations are.

One relation **OR** another is true whenever ***at least one*** of the two relations is true. (Remember that it is still true if ***both*** the relations are true - this is something that English doesn't always imply.)

The **NOT** relationship turns things upside down. The **NOT** relation is true whenever the relation is false, and false whenever it is true.

Logical expressions may use combinations of **AND OR** and **NOT**, just as numerical expressions may use combinations of +, -, * and so on. You can even put them in brackets if necessary. Logical operations have priorities in the same way as + - * / and ↑ do. **NOT** has the highest priority, then **AND**, then **OR**.

NOT is really a function, with an argument and a result, but its priority is much lower than that of other functions. Therefore, its argument does not need brackets unless it contains **AND** or **OR** (or both). **NOT a=b** means the same as **NOT (a=b)** (and the same as **a<>b** of course).

<> is the negation of = in the sense that it is true only if = is false. In other words...

a<>b is the same as **NOT a=b**

...and also

NOT a<>b is the same as **a=b**

Convince yourself that >= and <= are the negations of < and > respectively. Thus you can always get rid of **NOT** from in front of a relation by changing the relation.

Also..

NOT (a first logical expression **AND** a second)

..is the same as..

NOT (the first) **OR** **NOT** (the second)

..and..

NOT (a first logical expression **OR** a second)

..is the same as..

NOT (the first) **AND** **NOT** (the second)

Using this, you can work **NOT**s through brackets until eventually they are all applied to relations, and then you can get rid of them. Logically speaking, **NOT** is unnecessary, although you might still find that using it makes a program clearer.

'The following section is quite complicated, and can be skipped by the faint-hearted'

Try..

```
PRINT 1=2,1 <> 2
```

which you might expect to give a syntax error. In fact, as far as the computer is concerned, there is no such thing as a logical value - instead it uses ordinary numbers, subject to a few rules..

(i) =, <, >, <=, >= and <> all give the numeric results 1 for true, and 0 for false. Thus, the **PRINT** command above printed 0 for 1=2, which is false, and 1 for 1<>2, which is true.

(ii) In the statement

IF condition **THEN**

..the condition can be actually any numeric expression. If its value is 0, then it counts as false, and any other value (including the value of 1 that a true relation gives) counts as true. Thus the **IF** statement means exactly the same as

IF condition <> **Ø THEN**

(iii) **AND**, **OR** and **NOT** are also number-valued operations..

x AND y has the value { x, if y is true (non-zero); 0 (false), if y is false (zero) }

x OR y has the value { 1 (true), if y is true (non-zero); x, if y is false (zero) }

NOT x has the value { 0 (false), if x is true (non-zero); 1 (true), if x is false (zero) }

(Notice that 'true' means non-zero when we're checking a given value, but it means 1 when we're producing a new one)

Now try this program..

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a >= b)+(b AND
    a< b)
40 GO TO 10
```

Each time it prints the larger of the two numbers **a** and b

Convince yourself that you can think of

```
x AND y
```

..as meaning.

x if y (else the result is 0)

and of

```
x OR y
```

as meaning

x unless y (in which case the result is 1)

An expression using **AND** or **OR** like this is called a conditional expression. An example using **OR** could be

```
LET total=price less tax*(1.15 OR v$="zero rated")
```

Notice how **AND** tends to go with addition (because its default value is 0) and **OR** tends to go with multiplication (because its default value is 1)

You can also make string valued conditional expressions, but only using **AND**.

x$ AND y has the value { **x$** if y is non-zero; "" if y is zero }

.so it means **x$** if y (else the empty string)

Try this program which inputs two strings and puts them in alphabetical order

```
10 INPUT "type in two strings"
    'a$,b$
20 IF a$> b$ THEN LET c$=a$: L
   ET a$=b$: LET b$=c$
30 PRINT a$;" ";("<" AND a$< b
   $)+("=" AND a$=b$);" ";b$
40 GO TO 10
```

Part 14
The Character Set

Subjects covered...

```
CODE, CHR$
POKE, PEEK
USR
BIN
```

The letters, digits, spaces, punctuation marks and so on that can appear in strings are called characters, and they make up the character set that the *+3* uses. Most of these characters are single symbols, but there are some more, called tokens, that represent whole words, such as **PRINT**, **STOP**, **<>** and so on.

There are 256 characters, and each one has a code between 0 and 255 (there is a complete list of them in part 28 of this chapter). To convert between codes and characters, there are two functions, **CODE** and **CHR$**.

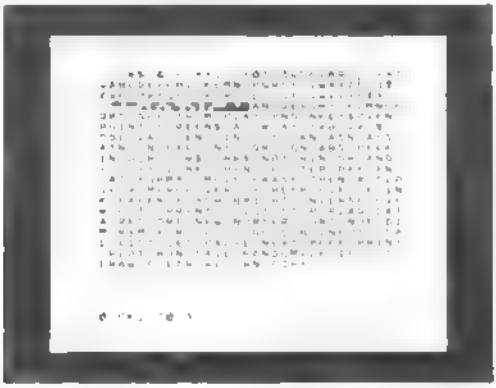
CODE is applied to a string, and gives the code of the first character in the string (or 0 if the string is empty).

CHR$ is applied to a number, and gives the single character string whose code is that number.

This program prints out the entire character set:

```
10 FOR a=32 TO 255: PRINT CHR$
   a;: NEXT a
```

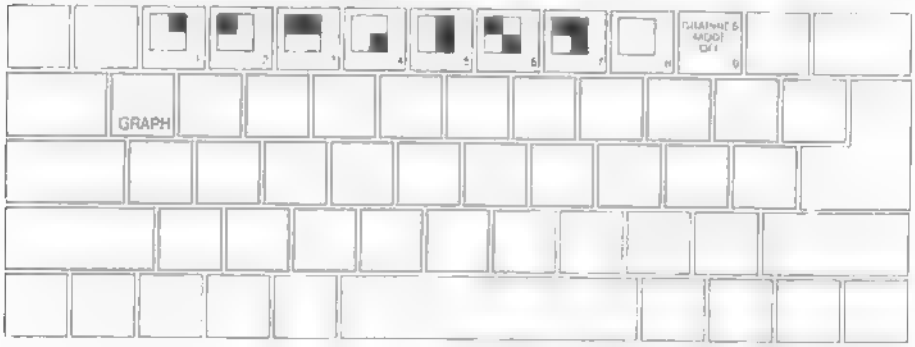
On the screen will appear the following:



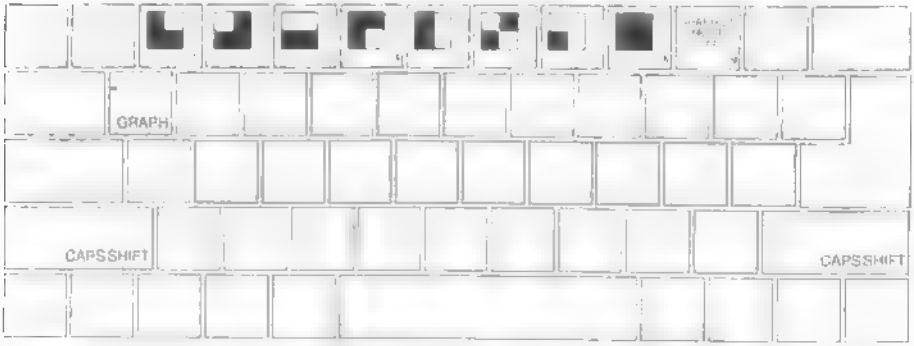
The character set

As you can see, the character set consists of a space, 15 symbols and punctuation marks, the ten digits, seven more symbols, the capital letters, six more symbols, the lower case letters and five more symbols. These are all (except £ and ©) taken from a widely-used set of characters known as ***ASCII*** (American Standard Codes for Information Interchange). ASCII also assigns numeric codes to these characters, and these are the codes that the ***+3*** uses.

The rest of the characters are not part of ASCII, but are dedicated to the ZX Spectrum range of computers. First amongst them are a space and 15 patterns of black and white blobs. These are called the ***graphics symbols*** and can be used for drawing pictures. You can enter these from the keyboard, using what's known as ***graphics mode***. Pressing the **GRAPH** key switches on graphics mode, after which the keys **1 2 3 4 5 6 7** and **8** will produce the graphics symbols.



While in graphics mode, pressing **CAPS SHIFT** together with one of the keys **1** to **8** produces 'inverted' versions of the same symbols, ie. black becomes white and white becomes black.



The cursor keys won't work properly while all this is going on as the ***+3*** interprets them as shifted number keys, and prints graphics characters accordingly.

Pressing the **9** key turns everything back to normal (as does pressing **GRAPH** again). The **0** key deletes the character to the left of the cursor.

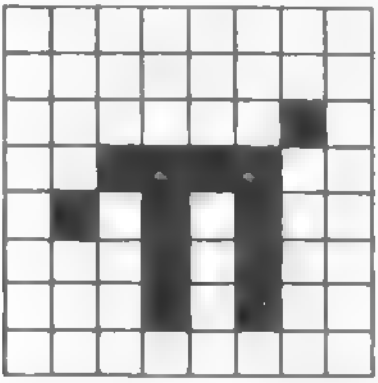
Here are the sixteen graphics symbols...

Symbol	Code	Symbol	Code
	128		143
	129		142
	130		141
	131		140
	132		139
	133		138
	134		137
	135		136

After the graphics symbols in the character set, you will see what appears to be another copy of the alphabet from A to S. These are characters that you can redefine yourself (though when the machine is first switched on they are set as letters) - they are called ***user-defined graphics***. You can type these in from the keyboard by going into graphics mode, and then using the letter keys **A** to **S**.

To define a new character for yourself, follow this recipe - it defines a character to show π.

(i) Work out what the character looks like. Each character has an 8 x 8 grid of dots, each of which can appear to be either on or off. You'd draw a diagram something like this (with black squares representing the dots which are on).



When a dot ■ on, the ***+3*** prints the ink colour; when a dot ■ off, the ***+3*** prints the paper colour. (The terms ink and paper are explained in part 16 of this chapter.)

We've left a one-square border around the edge of the character because all the other letters also have one (except for lower case letters with tails, where the tail goes right down to the bottom).

(ii) Work out which user-defined graphic you wish to display π - let's say the one corresponding to **P**, so that if you press **P** (after pressing **GRAPH**) you get π.

(iii) Store the new pattern. Each user-defined graphic has its pattern stored as eight numbers, one for each row. You can write each of these numbers ■ a program as **BIN** followed by eight 0's or 1's - 0 for paper, 1 for ink - so the eight numbers for our π character are:

```
BIN 00000000   - top row
BIN 00000000   - second row down
BIN 00000010   - third row down
BIN 00111100   - fourth row down
BIN 01010100   - fifth row down
BIN 00010100   - sixth row down
BIN 00010100   - seventh row down
BIN 00000000   - bottom row
```

(If you know about binary numbers, then it should help you ■ know that **BIN** is used to write a number in binary instead of the usual decimal.) Look at the pattern of binary numbers through half-closed eyes - you may even be able to see the π character.

These eight numbers are stored in eight locations (bytes) in memory. Each of these locations has an ***address***. The address of the first byte (or group of eight digits) is **USR "P"** (we chose P in (ii) above). The address of the second byte is **USR "P"+1**, and so on up to the eighth byte, which has the address **USR "P"+7**.

USR here is a function to convert a string argument into the address of the first byte in memory for the corresponding user-defined graphic. The string argument must be a single character which can be either the user-defined graphic itself or the corresponding letter (in upper or lower case). There is another use for USR when its argument is a number, which will be dealt with later.

Even if you don't understand this, the following program will define the character for you.

```
 10 FOR n=0 TO 7
 20 READ row: POKE USR "P"+n, r
    ow
 30 NEXT n
 40 DATA BIN 00000000
 50 DATA BIN 00000000
 60 DATA BIN 00000010
 70 DATA BIN 00111100
 80 DATA BIN 01010100
 90 DATA BIN 00010100
100 DATA BIN 00010100
110 DATA BIN 00000000
```

The POKE statement stores a number directly in a memory location, bypassing the mechanisms normally used by the BASIC. The opposite of POKE is PEEK, and this allows us to look at the contents of a memory location although it does not actually alter the contents themselves. PEEK and POKE are described more fully in part 24 of this chapter.

After the user-defined graphics in the character set come the *tokens*.

You will have noticed that we have not printed out the first 32 characters (codes 0 to 31) - these are *control characters*. They don't produce anything printable, but instead are used to control the screen display or some other function of the *+3*.

(If you try to print control characters, the *+3* displays ? to show that it doesn't understand them. Control characters are described more fully in part 28 of this chapter.)

The three control characters that the screen display uses are 6, 8 and 13 (these will now be explained). On the whole, CHR$ 8 is the only one you are likely to find useful.

CHR$ 6 prints spaces in exactly the same way as a comma does in a PRINT statement, for instance:

```
PRINT 1; CHR$ 6;2
```

does the same as

```
PRINT 1,2
```

Obviously this is not a very clear way of using it. A more subtle way is to say:

```
LET a$="1"+ CHR$ 6+"2"
PRINT a$
```

CHR$ 8 is 'backspace' - it moves the print position back one place. Try.

```
PRINT "1234"; CHR$ 8;"5"
```

which prints out.

```
1235
```

CHR$ 13 is 'newline' - it moves the print position to the beginning of the next line.

The screen display also uses control codes 16 to 23 - these are explained in parts 15 and 16 of this chapter (all the codes are listed in part 28).

Using the codes for the characters we can extend the concept of 'alphanumerical ordering' to cover strings containing any characters, not just letters. If instead of thinking in terms of the usual alphabet of 26 letters we use the extended alphabet of 256 characters, in the same order as their codes, then the principle is exactly the same. For instance, the following strings are in their 'Spectrum' ASCII alphabetical order. (Notice the rather odd feature that lower case letters come after all the capitals, so a comes after Z. Notice also that spaces are significant.)

```
CHR$ 3+"ZOOLOGICAL GARDENS"
CHR$ 8+"AARDVARK HUNTING"
" AAAARGH!"
"(Parenthetical remark)"
"100"
"129.95 inc. VAT"
"AASVOGEL"
"Aardvark"
"Elgar, the Regal Lager"
"PRINT"
"Zoo"
"[interpolation]"
"aardvark"
"aasvogel"
"derby"
"zoo"
"zoology"
```

Here is the rule for finding out in which order two strings come. Start by comparing the first two characters. If they are different, then one of them has its code less than the other, and the string it comes from is the earlier (lesser) of the two strings. If they are the same, then go on to compare the next two characters. If in this process one of the strings runs out before the other, then that string is the earlier; otherwise they must be equal.

The relations =, <, >, <=, >=, and <> are used for strings as well as for numbers. < means 'comes before' and > means 'comes after', so that.

```
"AA man"<"AARDVARK"
"AARDVARK">"AA man"
```

...are both true.

<= and >= work the same way as they do for numbers, so that..

```
"The same string" <= "The same string"
```

...is true, but..

```
"The same string" < "The same string"
```

...is false.

Experiment on all this using the program here, which inputs two strings and puts them in order.

```
10 INPUT "Type in two strings:
   ",a$,b$
20 IF a$> b$ THEN LET c$=a$: L
   ET a$=b$: LET b$=c$
30 PRINT a$;" ";
40 IF a$< b$ THEN PRINT "<";:
   GO TO 60
50 PRINT "=";
60 PRINT " ";b$
70 GO TO 10
```

Note (in the above program and also in the program at the end of part 13) how we have to introduce **c$** in line 20 when we swap over **a$** and **b$**. Can you see why simply using

```
LET a$=b$: LET b$=a$
```

would not have the desired effect?

The next program sets up user defined graphics for the following keys to display chess pieces.

B for bishop
K for king
R for rook
Q for queen
P for pawn
N for knight

Chess pieces

```
 5 LET b=BIN 01111100: LET c=B
   IN 00111000: LET d=BIN 0001
   0000
10 FOR n=1 TO 6: READ p$: REM
   6 pieces
20 FOR f=0 TO 7: REM read piec
   es into 8 bytes
```

```
 30 READ a: POKE USR p$+f, a
 40 NEXT f
 50 NEXT n
100 REM bishop
110 DATA "b", 0, d, BIN 0010100
    0, BIN 01000100
120 DATA BIN 01101100, c, b, 0
130 REM king
140 DATA "k", 0, d, c, d
150 DATA c, BIN 01000100, c, 0
160 REM rook
170 DATA "r", 0, BIN 01010100,
    b, c
180 DATA c, b, b, 0
190 REM queen
200 DATA "q", 0, BIN 01010100,
    BIN 00101000, d
210 DATA BIN 01101100, b, b, 0
220 REM pawn
230 DATA "p", 0, 0, d, c
240 DATA c, d, b, 0
250 REM knight
260 DATA "n", 0, d, c, BIN 0111
    1000
270 DATA BIN 00011000, c, b, 0
```

Note that in the above **DATA** statements we have simply used **0** instead of **BIN 00000000**

When you have run this program you may look at the pieces by pressing **GRAPH** followed by any of the keys **B K R Q P or N**

Exercises

1. Imagine the space [illegible] symbol [illegible] quarter-[illegible] character cake. Then if each quarter can be either black or white there are [illegible] possibilities. Find the [illegible] to the character set.

2. Run this program

```
10 INPUT c
20 PRINT CHR$ c;
30 GO TO 10
```

If you experiment with it you will find that **CHR$** c is rounded to the nearest whole number, and if c is not in the range [illegible] then the program stops with the error report **B integer out of range**

3. Which of these is the lesser?

```
"EVIL"
"evil"
```

Part 15
More about PRINT and INPUT

Subjects covered...

- CLS
- PRINT items
- Expressions (numeric or string type)
- TAB numeric expression
- AT numeric expression
- PRINT separators , ; '
- INPUT items
- Variables (numeric or string type)
- LINE string variable
- Scrolling
- SCREEN$

You have already seen PRINT used quite a lot so you will have a rough idea of how it is used. Expressions whose values are printed are called PRINT *items*. They may be separated by commas, semicolons or apostrophes, which are called PRINT *separators*. A PRINT item can also be nothing at all, which is a way of explaining what happens when you use PRINT on its own.

There are two more kinds of PRINT items, which are used to tell the *+3* not what, but where to print. For example, the instruction.

```
10 PRINT AT 11,16;"*"
```

...prints an asterix * in the centre of the screen. This is because

AT line,column

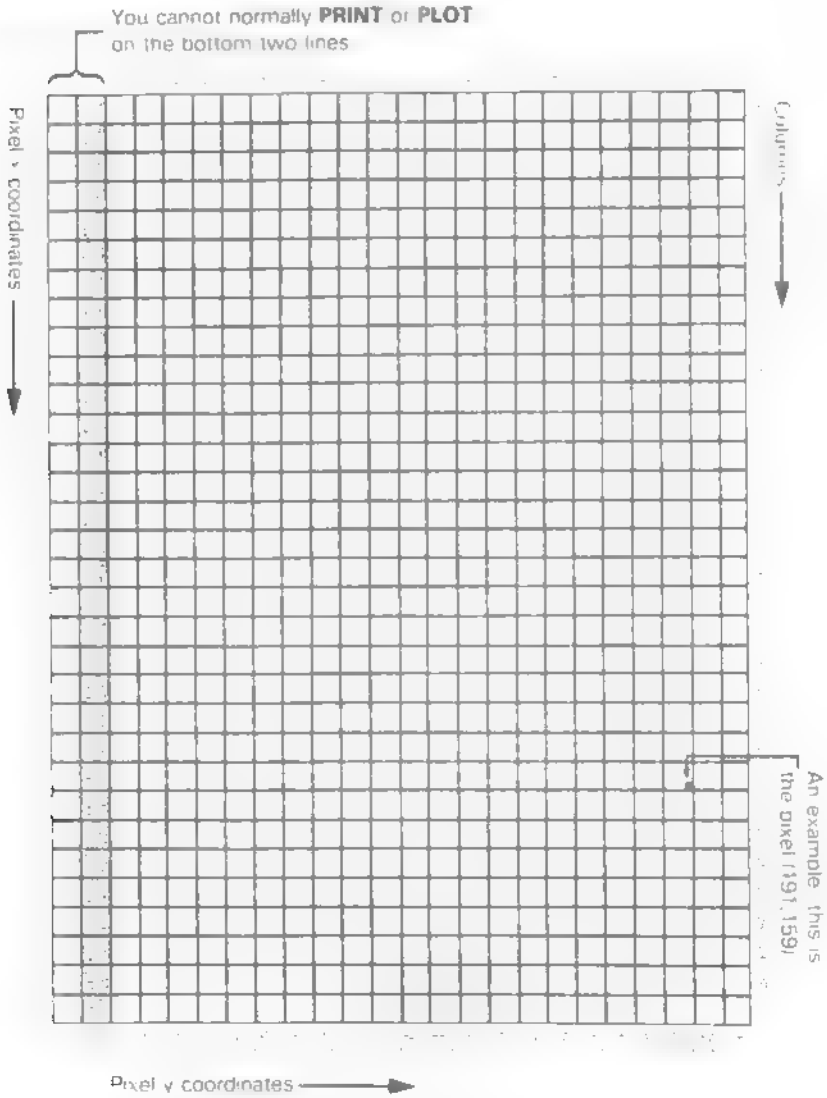
moves the PRINT position (the place where the next item is to be printed) to the line and column specified. Lines are numbered from 0 (at the top) to 21, columns are numbered from 0 (on the left) to 31.

SCREEN$ is the reverse function to PRINT AT, and will (within limits) 'read' the character which is located at a particular position on the screen. It uses line and column numbers in the same way as PRINT AT, but enclosed in brackets. For example, the instruction.

```
20 PRINT AT 0,0; SCREEN$ (11,16)
```

, will read the asterix printed in the centre of the screen, then print it in location 0,0 (the top left-hand corner).

Characters from tokens are read normally (as single characters), and spaces are read as spaces. However, attempting to read user-defined characters, graphics characters, or lines drawn by **PLOT**, **DRAW** and **CIRCLE**, result in a null (empty) string being returned. The same applies if **OVER** has been used to create a composite character. (The keywords **PLOT**, **DRAW**, **CIRCLE** and **OVER** are described in parts 16 and 17 of this chapter.)



The function..

```
TAB column
```

...prints enough spaces to move the **PRINT** position to the column specified. It stays on the same line, or if this would involve backspacing, moves to the next line. Note that the ***+3*** reduces the column number 'modulo 32' (ie it divides by 32 and takes the remainder) - so **TAB 33** means the same as **TAB 1**

As an example

```
PRINT TAB 30;1; TAB 12;"Contents
"; AT 3,1;"Chapter"; TAB 24;"Pag
e"
```

..is how you might want to print out the heading on the contents page (page 1) of a book

Try running this.

```
10 FOR n=0 TO 20
20 PRINT TAB 8*n;n;
30 NEXT n
```

This shows what is meant by the **TAB** numbers being reduced modulo 32

For a more elegant example, change the 8 in line [illegible] a 6

Note the following points

(i) **TAB**s and print items are best terminated with semicolons as we have done above. You can use commas (or nothing, at the end of the statement), but this means that after having carefully set up the **PRINT** position, you immediately move it on again - not terribly useful!

(ii) You cannot print on the bottom two lines (22 and 23) on the screen because they are reserved for commands, **INPUT** data, reports, error messages and so on. References to 'the bottom line' usually mean line 21

(iii) You can use **AT** to locate the **PRINT** position even where there is already something printed -the new print item will simply overwrite the old

Another statement connected with **PRINT** is **CLS**. This clears the whole screen.

When printing reaches the bottom of the screen, it starts to scroll upwards rather like a typewriter. You can see this if you go into the small screen using the edit menu option **Screen** (described in chapter 5), and then type..

```
CLS: FOR n=1 TO 30: PRINT n: NEXT n
```

When it has printed a screen full, the ***+3*** will stop with the message **scroll?** at the bottom of the screen. You can now inspect the first 22 numbers at your leisure. When you have finished with them,

press **Y** (for yes) and the *+3* will give you the next screen full of numbers. Actually, any key will make the *+3* carry on except **N** (for no), the **BREAK** key or the space bar. These will make the *+3* stop running the program with the report **D BREAK - CONT** repeats.

The **INPUT** statement can do much more than we have told you so far. You have already seen **INPUT** statements like..

```
INPUT "How old are you?", age
```

in which the *+3* prints the caption **How old are you?** at the bottom of the screen, and then you have to type in your age. In fact though, an **INPUT** statement can be made up of items and separators in exactly the same way as a **PRINT** statement so **How old are you?** and **age** are both **INPUT** items. **INPUT** items are generally the same as **PRINT** items, however there are some very important differences.

First, an obvious extra **INPUT** item is the variable whose value you require to be typed in - **age** in our example above. The rule is that if an **INPUT** item begins with a letter, then it must be a variable whose value is to be input.

This would seem to mean that you can't print out the values of variables as part of a caption. However, you can get round this by putting brackets around the variable. Any expression that starts with a letter must be enclosed in brackets if it is to be printed as part of a caption.

Any kind of **PRINT** item that is not affected by these rules is also an **INPUT** item. Here is an example to illustrate what's going on.

```
LET my age = INT ( RND * 100): I
NPUT ("I am ";my age;".");" How
old are you?", your age
```

my age is contained in brackets, so its value gets printed out. **your age** is not contained in brackets, so you have to type its value in.

Everything that an **INPUT** statement writes goes to the bottom part of the screen, which acts somewhat independently of the top part. In particular, its lines are numbered relative to the top line of the bottom half, even if this has moved up the actual TV screen (which it does if you type lots of **INPUT** data). Whatever the small screen does during **INPUT**, however, it will always revert to being two lines in size when the program stops, and you start editing.

To see how **AT** works in **INPUT** statements try this.

```
10 INPUT "This is line 1.",a$;
    AT 0,0;"This is line 0.",a
   $; AT 2,0;"This is line 2."
   ,a$; AT 1,0;"This is still
   line 1.",a$
```

Run the program (just press **ENTER** each time it stops). When **This is line 2** is printed, the lower part of the screen moves up to make room for it, but the numbering moves up as well, so that the lines of text keep their same numbers.

Now try this..

```
10 FOR n=0 TO 19: PRINT AT n, 0
   ;n;: NEXT n
20 INPUT AT 0, 0;a$; AT 1, 0;a$
   ; AT 2, 0;a$; AT 3, 0;a$; AT
    4, 0;a$; AT 5, 0;a$;
```

As the lower part of the screen goes up and up, the upper part remains undisturbed until the lower part threatens to write on the same line as the **PRINT** position. Then the upper part starts scrolling up to avoid this.

Another refinement to the **INPUT** statement that we haven't seen yet is called **LINE** input and is a different way of inputting string variables. If you use **LINE** before the name of a string variable to be input, as in :

```
INPUT LINE a$
```

then the ***+3*** will not give you the string quotes that it normally does for a string variable (though it will pretend to itself that they are there). So if you type in

```
bugs
```

as the **INPUT** data, **a$** will be given the value **bugs**. Because the string quotes do not appear with the string, you cannot delete them and type in a different sort of string expression for the **INPUT** data. Remember that you cannot use **LINE** for numeric variables.

There's an interesting side effect to **INPUT**. Whilst typing into an **INPUT** request, the old Spectrum single-key entry system enjoys a brief moment of freedom before being locked away again when you press **ENTER**. Run this program if you're interested.

```
10 INPUT numbers
20 PRINT numbers
30 GO TO 10
```

Input a few numbers, and they'll be printed faithfully onto the screen. Now press **EXTEND MODE** followed by the **M** key. The word **PI** appears, and if you press **ENTER**, then **3.1415927** will appear as if by magic. However, if you type **PI** as two letters without the aid of **EXTEND MODE** then the ***+3*** will stop with the report **2 Variable not found, 10:1**

There's no simple explanation for this behaviour, and it's best just to be aware that it can happen if you press some combinations of keys during **INPUT**. If for some reason you're keen to experiment, chapter 7 (Using 48 BASIC) will tell you which keys produce which effects.

The control characters **CHR$ 22** and **CHR$ 23** have effects rather like **AT** and **TAB**. Whenever the *+3* is instructed to print one of them the character must be followed by two more characters that do not have their usual effect, but that are treated instead as numbers (their codes) to specify the line and column (for **AT**) or the tab position (for **TAB**). You will almost always find it easier to use **AT** and **TAB** in the usual way rather than use control characters, however, they might be useful in some circumstances. The **AT** control character is **CHR$ 22**. The first character after it specifies the line number and the second specifies the column number, so that.

```
PRINT CHR$ 22+ CHR$ 1+ CHR$ c;
```

.. has exactly the same effect as.

```
PRINT AT 1, c;
```

This is so that even if **CHR$ 1** or **CHR$ c** would normally have a different meaning (for instance if **c=13**) the **CHR$ 22** before them overrides that.

The **TAB** control character is **CHR$ 23** and the two characters after it combine to give a number between 0 and 65535, specifying the number you would have in a **TAB** item. The statement..

```
PRINT CHR$ 23+ CHR$ a+ CHR$ b;
```

.. has the same effect as..

```
PRINT TAB a+256*b;
```

You can use **POKE** to stop the computer asking if you wish to **scroll?** by typing

```
POKE 23692, 255
```

..every so often. After this it will scroll up 255 times before stopping with **scroll?** As an example, try

```
10 FOR n=0 TO 1000
20 PRINT n: POKE 23692,255
30 NEXT n
```

.. and watch everything whizz off the screen!

Exercise.

1. Try this program on some children, to test their multiplication tables.

```
 10 LET m$=""
 20 LET a= INT ( RND *12)+1: LE
    T b = INT ( RND *12)+1
 30 INPUT (m$) ' ' "what is ";(
    a);" x ";(b);"?";c
100 IF c=a*b THEN LET m$="Right
    .": GO TO 20
110 LET m$="Wrong. Try again.":
    GO TO 30
```

If they are perceptive, they might manage to work out that they do not have to do the calculation themselves. For instance, if the **+3** asks them to type the answer to 2 x 3, then all they have to do is type in **2*3** literally.

Part 16
Colours

Subjects covered...

```
INK, PAPER, FLASH, BRIGHT, INVERSE, OVER
BORDER
```

Run this program

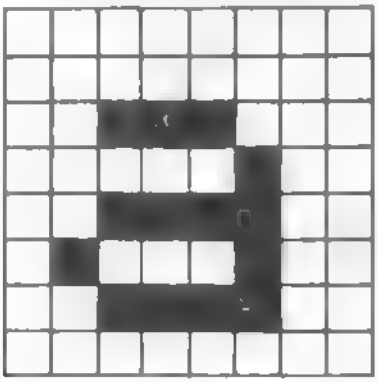
```
 10 FOR m=0 TO 1: BRIGHT m
 20 FOR n=1 TO 10
 30 FOR c=0 TO 7
 40 PAPER c: PRINT "    ";: REM
     4 coloured spaces
 50 NEXT c: NEXT n: NEXT m
 60 FOR m=0 TO 1: BRIGHT m: PAP
    ER 7
 70 FOR c=0 TO 3
 80 INK c: PRINT c;"  ";
 90 NEXT c: PAPER 0
100 FOR c=4 TO 7
110 INK c: PRINT c;"  ";
120 NEXT c: NEXT m
130 PAPER 7: INK 0: BRIGHT 0
```

This shows the eight colours matching white and black, and the two levels of brightness that the ***+3*** can produce on a colour TV. (If your TV is black-and-white then you will see just various shades of grey.) A quicker way to achieve a similar result is to **RESET** the ***+3*** whilst holding down **BREAK** but that's a little drastic. Here is a list of which numbers produce which colours (for your reference)

0 - black
1 - blue
2 - red
3 - magenta
4 - green
5 - cyan
6 - yellow
7 - white

On a black-and-white TV these numbers are in order of brightness. To use these colours properly you need to understand a bit about how the picture is arranged

The picture is divided up into 768 (24 lines of 32) positions (cells) where characters can be printed.



A typical character cell

Each character cell consists of an 8 x 8 grid (such as above). This should remind you of the user-defined graphics in part 14, where we had 0s for the white dots and 1s for the black dots.

The character has two colours associated with it: the ***ink***, or foreground colour, which is the colour for the black dots in our square, and the ***paper*** or background colour, which is used for the white dots. To start off with, every cell has black ink and white paper so writing appears as black on white.

The character also has a brightness (normal or extra bright), and something to say whether it flashes or not. Flashing is done by continuously swapping the ink and paper colours. All this information can be coded into numbers, so a character then has the following:

(i) An 8 x 8 grid of 0s and 1s to define the shape of the character, with 0 for paper and 1 for ink.

(ii) Ink and paper colours, each coded into a number between 0 and 7.

(iii) A brightness - 0 for normal, 1 for extra bright.

(iv) A flash number - 0 for steady, 1 for flashing.

Note that since the ink and paper colours cover a whole character cell, you cannot possibly have more than two colours in a given block of 64 dots. The same goes for the brightness and flash numbers - they refer to the whole character cell, not individual dots within the cell. The colour, brightness and flash number for a given character cell are called ***attributes***.

When you print something on the screen, you change the dot pattern for that character cell. It is less obvious, but still true, that you also change the cell's attributes. To start off with you do not notice this because everything is printed with black ink on white paper (at normal brightness and no flashing); however, you can vary this with the **INK**, **PAPER**, **BRIGHT** and **FLASH** statements. Using the edit menu's **Screen** option, go to the bottom screen, and try..

```
PAPER 5
```

...and then **PRINT** a few items on the screen - they will appear on cyan paper, because as they are printed, the paper colour for the cells they occupy are set to cyan (which has code 5)

The others work the same way, so you may use the settings...

PAPER	(whole number between 0 and 7)
INK	(whole number between 0 and 7)
BRIGHT	(whole number between 0 and 1)
FLASH	(whole number between 0 and 1)

...and any printing will set the corresponding attributes for all the character cells it subsequently uses

Try some of these out. You should now be able to see how the program at the beginning of this section worked (remember that a space is a character that has its ink and paper the same colour)

There are some more numbers you can use in these statements that have less direct effects.

8 can be used in all four statements, and means 'transparent' in the same sense that the old attribute shows through. Suppose, for instance, that you do

```
PAPER 8
```

No character position will ever have its paper colour set to 8 because there is no such colour; what happens is that when a position is printed on, its paper colour is left the same as it was before. However, **INK 8**, **BRIGHT 8** and **FLASH 8** work the same way as for the other attribute numbers.

9 can be used only with **PAPER** and **INK** and means 'contrast'. The colour (ink or paper) that you use it with is made to contrast with the other by being made white if the other is a dark colour (black, blue, red or magenta), or being made black if the other is a light colour (green, cyan, yellow or white).

Try this by doing

```
INK 9: FOR c=0 TO 7: PAPER c: PRINT c: NEXT c
```

A more impressive display of its power is to run the program at the beginning to make coloured stripes (again, making sure that you are in the lower screen when you type **RUN**), and then doing.

```
INK 9: PAPER 8: PRINT AT 0, 0;:
FOR n=1 TO 1000: PRINT n;: NEXT n
```

The ink colour here is always made to contrast with the old paper colour for each character cell.

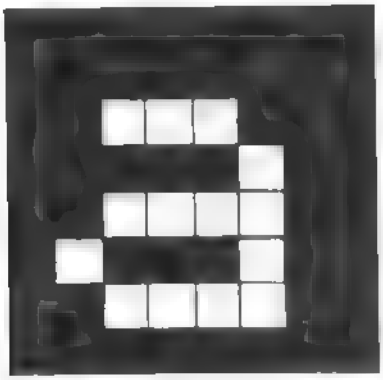
Colour TV relies on the fact that the human eye need see only three colours at a time (red, green and blue) in various combinations and intensities in order to perceive all the colours of the spectrum. The ***+3*** also displays its spectrum of colours by using mixtures of red, green and blue. For instance, yellow is made by mixing red with green – which is why its code, 6, is the sum of the codes for red and green.

To see how all eight colours fit together, imagine three rectangular spotlights, coloured red, green and blue, shining at not quite the same place on a piece of white paper in the dark. Where they overlap you will see mixtures of colours, as shown by the following program (note that solid ink spaces are obtained by entering graphics mode (pressing **GRAPH**, then holding down **CAPS SHIFT** while pressing **8**. To exit from graphics mode, press **9**.

```
   10 BORDER 0: PAPER 0: INK 7: C
      LS
   20 FOR a=1 TO 6
   30 PRINT TAB 6; INK 1;"
      ": REM 18 ink sq
      uares
   40 NEXT a
   50 LET dataline=200
   60 GO SUB 1000
   70 LET dataline=210
   80 GO SUB 1000
   90 STOP
  200 DATA 2,3,7,5,4
  210 DATA 2,2,6,4,4
 1000 FOR a=1 TO 6
 1010 RESTORE dataline
 1020 FOR b=1 TO 5
 1030 READ c: PRINT INK c;"
      ";: REM 6 ink squares
 1040 NEXT b: PRINT: NEXT a
 1050 RETURN
```

There is a function called **ATTR** that finds out what the attributes are at a given position on the screen. It is a fairly complicated function, so it has been relegated to the end of this section.

There are two more statements, **INVERSE** and **OVER**, which control not the attributes, but the dot pattern that is printed on the screen. They take the [illegible] for off, [illegible] 1 for on. If you use **INVERSE 1** then each character [illegible] the inverse of its usual form, ie paper dots will be replaced by ink dots and vice versa. Thus the character cell containing **a** (shown previously) would be printed as follows (on the next page)



If (as at switch on) we have black ink and white paper, then the **a** will appear as white on black.

The statement

```
OVER 1
```

sets interaction to particular screen overprinting. Normally when anything is written into a character position, it completely obliterates what was there before. However, using **OVER 1**, the new character is simply superimposed. This can be particularly useful for writing composite characters, for an underlined letter as in the following. Type the program into the computer and select **+3 BASIC**. Note that the underline character is obtained by pressing **SYMB SHIFT** together with **Ø**:

```
10 OVER 1
20 PRINT "w"; CHR$ 8;"_";
```

(Notice we have used the control character **CHR$ 8** (backspace) to overprinting the **w** with **_**.)

There is another way of using **INK**, **PAPER** and so on, which you will probably find more useful than having them as statements. You can put them as items in a **PRINT** statement followed by **;**, and they then do exactly the same as they would have done if they had been used as statements on their own, except that their effect is only temporary, lasting as far as the end of the **PRINT** statement that contains them. Thus if you type

```
PRINT PAPER 6;"x";: PRINT "y"
```

then only the **x** will be on yellow paper.

PAPER, INK etc when used as statements do not affect the colour in the bottom part of the screen (where INPUT data is typed in and reports are displayed). The bottom screen uses the colour of the border for its paper colour, code 9 (for contrast) for its ink colour, has flashing off, and everything at normal brightness. You can change the border colour to any of the eight normal colours (not 8 or 9) using the statement..

```
BORDER colour
```

When you type in INPUT data, it follows this rule of using contrasting ink on border coloured paper; but you can change the colour of the captions written by the +3 by using PAPER INK, etc items in the INPUT statement, just as you would in a PRINT statement. Their effect lasts either to the end of the statement, or until some INPUT data is typed in, whichever comes soonest. Try

```
INPUT FLASH 1; INK 4;"Enter a number?";n
```

The +3 has a high regard for your sanity - no matter what combination of effects and colours you manage to produce from a BASIC program, the editor will always use black ink on white paper.

There is one more way of changing the colours by using control characters - rather like the control characters for AT and TAB in part 8.

```
CHR$ 16  corresponds to  INK
CHR$ 17  corresponds to  PAPER
CHR$ 18  corresponds to  FLASH
CHR$ 19  corresponds to  BRIGHT
CHR$ 20  corresponds to  INVERSE
CHR$ 21  corresponds to  OVER
```

These are each followed by one character that shows a colour by its code, so that (for instance)

```
PRINT CHR$ 16+ CHR$ 9;"item"
```

..has the same effect as

```
PRINT INK 9;"item"
```

On the whole, you would not bother to use these control characters because you might just as well use the statements PAPER, INK etc. However, if you have some old 48K BASIC programs on cassette, you may find such control characters embedded in the listing. In general, the editor will actively ignore them and remove them at the first opportunity. It is not possible to insert them into listings as with the old 48K Spectrum.

The ATTR function has the form

```
ATTR (line ,column)
```

Its two arguments are the line and column numbers that you would use in an **AT** item, and its result is a number that shows the colours and so on at the corresponding character position on the TV screen. You can use this as freely in expressions as you can any other function.

The number that is the result is the sum of four other numbers as follows:

128	if the character cell is flashing, 0 if it is steady
64	if the character cell is bright, 0 if it is normal
8	multiplied by the code for the paper colour
1	multiplied by the code for the ink colour

For instance, if the character cell is flashing, normal brightness, yellow paper and blue ink, then the four numbers that we have to add together are 128, 0, 3x6=48 and 1, making 177 altogether. Test this with:

```
PRINT AT 0,0; FLASH 1; PAPER 6;
 INK 1;" "; ATTR (0,0)
```

Exercises

1. Try:

```
PRINT "B"; CHR$ 8; OVER 1;"/";
```

Where the / has cut through the B, it has left a white dot. This is the [illegible] overprinting works on the **+3**: two papers or two inks give a paper, one of each [illegible]. This has the interesting property that if [illegible] with the same thing twice, you end up with what you had at the beginning. If you now [illegible]:

```
PRINT CHR$ 8; OVER 1;"/"
```

Why do you [illegible] B

2. Run this program:

```
10 POKE 22527+ RND *704, RND *
   127
20 GO TO 10
```

[illegible] This program is changing the colours of squares on the TV [illegible] RND [illegible] that [illegible] randomly. The [illegible] that you [illegible] the hidden part [illegible] RND is pseudo-random, instead of truly [illegible]

Part 17
Graphics

Subjects covered...

PLOT, DRAW, CIRCLE
Pixels

For all of this section, type in the example programs, commands and **RUN** in the small screen (use the edit menu's **Screen** option)

In this section we shall see how to draw pictures on the **+3**. The part of the screen you can use has 22 lines and 32 columns, making 22x32=704 character positions. As you may remember from part 16 each of these character positions is made up of an 8 x 8 grid of dots which are called ***pixels*** (picture elements).

A pixel is specified by two numbers - its coordinates. The first, its x coordinate, says how far it is across from the extreme left-hand column. The second, its y coordinate, says how far it is up from the bottom. These coordinates are usually written as a pair in brackets, so (0,0) (225,0) (0,175) and (255,175) are the bottom left, bottom right, top left and top right corners of the screen.

If you have trouble memorising which coordinate is which, simply remember that ***x is a cross*** (x is across)

The statement

PLOT x coordinate **,** y coordinate

inks in the pixel with these coordinates, so this measles program

```
10 PLOT INT ( RND *256), INT (
   RND *176): INPUT a$: GO TO
   10
```

plots a random point each time you press **ENTER**

Here is a rather more interesting program. It plots a graph of the function **SIN** (a sine wave) for values between 0 and 2π.

```
10 FOR n=0 TO 255
20 PLOT n, 88+80* SIN (n/128*
   PI )
30 NEXT n
```

This next program plots a graph of **SQR** (part of a parabola) between 0 and 4.

```
10 FOR n=0 TO 255
20 PLOT n, 80* SQR (n/64)
30 NEXT n
```

Notice that pixel coordinates are rather different from the line and column in an **AT** item. You may find that the diagram in part 15 of this chapter is useful when working out pixel coordinates and line and column numbers.

To help you with your pictures, the **+3** will draw straight lines, circles and parts of circles for you, using the **DRAW** and **CIRCLE** statements.

The statement **DRAW** (to draw a straight line) takes the form:

```
DRAW x,y
```

The starting place of the line is the pixel where the last **PLOT**, **DRAW** or **CIRCLE** statement left off (this is called the **PLOT** position - **RUN**, **CLEAR**, **CLS** and **NEW** reset it to the bottom left-hand corner at 0,0), the finishing place of the line is x pixels to the right of that and y pixels up. The **DRAW** statement on its own determines the length and direction of the line, but not its starting point.

Experiment with a few **PLOT** and **DRAW** commands, for instance:

```
PLOT 0,100: DRAW 80,-35
PLOT 90,150: DRAW 80,-35
```

Notice that the numbers in a **DRAW** statement can be negative, but those in a **PLOT** statement can't.

You can also plot and draw in colour, although you have to bear in mind that colours always cover the whole of a character cell and cannot be specified for individual pixels. When a pixel is plotted, it is set to show the full ink colour, and the whole of the character cell containing it is given the current ink colour. This program demonstrates that point.

```
10 BORDER 0: PAPER 0: INK 7: C
   LS: REM black out screen
20 LET x1=0: LET y1=0: REM sta
   rt of line
30 LET c=1: REM for ink colour
   , starting blue
40 LET x2= INT ( RND *256): LE
   T y2= INT ( RND *176): REM
   random finish on line
50 DRAW INK c;x2-x1,y2-y1
60 LET x1=x2: LET y1=y2: REM n
   ext line starts where last
   one finished
70 LET c=c+1: IF c=8 THEN LET
   c=1: REM new colour
80 GO TO 40
```

The lines seem to get broader as the program goes on, and this is because a line changes the colours of all the inked-in pixels of all the character cells that it passes through. Note that you can embed **PAPER**, **INK** **FLASH** **BRIGHT** **INVERSE** and **OVER** items in a **PLOT** or **DRAW** statement just as you could with **PRINT** and **INPUT**. They go between the keyword and the coordinates, and are terminated by either semicolons or commas.

An extra frill with **DRAW** is that you can use it to draw parts of circles instead of straight lines, by including an extra number to specify an angle to be turned through. The form is :

```
DRAW x,y,a
```

x and y are used to specify the finishing point of the line just as before, and a is the number of radians that it must turn through as it goes. If a is positive then it turns to the left, if a is negative then it turns to the right. Another way of seeing a is as showing the fraction of a complete circle that will be drawn, (a complete circle is 2π radians) so if a equals π it will draw a semicircle, if a equals $0.5\times\pi$ a quarter of a circle, and so on.

For instance, suppose a equals π. Then whatever values x and y take, a semicircle will be drawn. Try.

```
10 PLOT 100,100: DRAW 50,50, PI
```

which will draw this.

The drawing starts off in a south-easterly direction, but by the time it stops, it is going north-west. In between, it has turned through 180 degrees, or π radians (the value of a).

Run the program several times, with **PI** replaced by various other expressions, eg. **-PI**, **PI/2**, **3*PI/2**, **PI/4**, **1.0**, etc.

The last statement in this section is **CIRCLE**, which draws an entire circle. You specify the coordinates of the centre and the radius of the circle using

CIRCLE x coordinate , y coordinate , radius

Just as with **PLOT** and **DRAW** you can put the various sorts of colour items in at the beginning of a **CIRCLE** statement

The **POINT** function tells you whether a pixel is ink or paper colour. Its two arguments are the coordinates of the pixel (which must be enclosed in brackets) and its result is 0 if the pixel is paper colour, or 1 if it is ink colour. Try

```
CLS : PRINT POINT (0,0): PLOT 0,
0: PRINT POINT (0,0)
```

Type

```
PAPER 7: INK 0
```

and investigate how **INVERSE** and **OVER** work inside a **PLOT** statement. These two affect just the relevant pixel, and not the rest of the character cell. They are normally off (0) in a **PLOT** statement, so you only need to mention them to turn them on (1)

Here is a list of the possibilities for reference

`PLOT;`	This is the usual form. It plots an ink dot, ie. sets the pixel to show the ink colour
`PLOT INVERSE 1;`	This plots a dot of 'ink eradicator', ie. it sets the pixel to show the paper colour
`PLOT OVER 1;`	This exchanges the pixel colour with whatever it was before, so if it was ink colour then it becomes paper colour, and vice versa
`PLOT INVERSE 1; OVER 1;`	This leaves the pixel exactly as it was before, but note that it also changes the **PLOT** position, so you might use it simply to do that

As another example of using the **OVER** statement, fill the screen up with writing using black on white, and then type

```
PLOT 0,0: DRAW OVER 1;255,175
```

This will draw a fairly decent line, even though it has gaps in it wherever it hits some writing. Now type in exactly the same command again. The line will vanish without leaving any trace whatsoever - this is the great advantage of **OVER 1**. If you had drawn the line using

```
PLOT 0,0: DRAW 255,175
```

, and erased it using

```
PLOT 0,0: DRAW INVERSE 1;255,175
```

then you would also have erased some of the writing.

Now try...

```
PLOT 0,0: DRAW OVER 1;250,175
```

...and try to 'undraw' it using...

```
DRAW OVER 1;-250,-175
```

This doesn't quite work because the pixels that the line uses on the way back are not quite the same as the ones that it used on the way there. You must therefore undraw a line in exactly the same direction as you drew it.

One way to get unusual colours is to speckle two normal ones together in a single square, using a user-defined graphic. Try this program.

```
1000 FOR n=0 TO 6 STEP 2
1010 POKE USR "a"+n, BIN 0101010
     1: POKE USR "a"+n+1, BIN 10
     101010
1020 NEXT n
1030 REM now press GRAPH then A
```

which gives the user-defined graphic corresponding to a chessboard pattern. If you print this character (press **GRAPH** then **A**) you will find that the character is reproduced in a combination of the current paper and ink colours.

Exercises:

1. Experiment with **PAPER INK FLASH** and **BRIGHT** items in a **PLOT** statement. These are the parts that affect the whole of the character cell containing the pixel. Normally it is as though the **PLOT** statement had started off:

```
PLOT PAPER 8; FLASH 8; BRIGHT 8;   etc
```

...and only the ink colour of a character cell is altered when something is plotted there, but you can change this if you wish.

Be especially careful when using colours with **INVERSE 1**, because this sets the pixel to show the paper colour, and may change the ink colour, which might not be what you expect.

2. If you have read part 10, see if you can work out how to draw circles using **SIN** and **COS**. Run this program:

```
10 FOR n=0 TO 2* PI STEP PI /1
   80
20 PLOT 100+80* COS n, 87+80*
   SIN n
30 NEXT n
40 CIRCLE 150, 87, 80
```

You can see that the **CIRCLE** statement is much quicker, albeit less accurate.

3. Try..

```
CIRCLE 100,87,80: DRAW 50,50
```

You can see from this that the **CIRCLE** statement leaves the **PLOT** position at a rather indeterminate place - it is always somewhere about half way up the right-hand side of the circle. You will usually need to follow the **CIRCLE** statement with a **PLOT** statement before you do any more drawing.

Part 18
Timing

Subjects covered...

PAUSE, PEEK, INKEY$

Quite often you will want to make the program take a specified length of time and for this you will find the PAUSE statement useful

```
PAUSE n
```

stops computing and displays the picture for n frames of the TV (there are 50 frames per second in Europe and 60 in USA) The value of n can be up to 65535 which gives you a pause of just under 22 minutes If n=0 then it means 'pause indefinitely'

A pause can always be cut short by pressing a key

This program works the second hand of a clock

```
 10 REM first we draw the clock
     face
 20 FOR n=1 TO 12
 30 PRINT AT 10-10* COS (n/6* P
    I ),16+10* SIN (n/6* PI );n
 40 NEXT n
 50 REM now we start the clock
 60 FOR t=0 TO 200000: REM t is
     the time in seconds
 70 LET a=t/30* PI : REM a is t
    he angle of the second hand
     in radians
 80 LET sx=80* SIN a: LET sy=80
    * COS a
200 PLOT 128,88: DRAW OVER 1;sx
    ,sy: REM draw second hand
210 PAUSE 42
220 PLOT 128,88: DRAW OVER 1;sx
    ,sy: REM erase second hand
400 NEXT t
```

The clock will run down after about [illegible] hours because of line 50 but you can easily make it run longer Note how the timing is controlled by line 210 You might expect PAUSE 50 to make it tick once per second, however the computing takes a bit of time as well and has to be allowed for This is best done by trial and error timing the +3 clock against a real one and adjusting line 210 until they agree You can't do this very accurately - an adjustment of one frame per second is equal to 2% (or half an hour in a day)

There is a much more accurate way of measuring time. This uses the contents of certain memory locations. The data stored is retrieved by using **PEEK**. Part 25 of this chapter explains what we're looking at in detail. Type in the expression.

```
PRINT (65536* PEEK 23674+256* PE
EK 23673+ PEEK 23672)/50
```

This prints the number of seconds since the ***+3*** was turned on (up to about 3 days and 21 hours, after which it goes back to 0).

Here is a revised clock program to make use of this.

```
 10 REM first we draw the clock
     face
 20 FOR n=1 TO 12
 30 PRINT AT 10-10* COS (n/6* P
    I ),16+10* SIN (n/6* PI );n
 40 NEXT n
 50 DEF FN t()= INT ((65536* PE
    EK 23674+256* PEEK 23673+ P
    EEK 23672)/50): REM number
    of seconds since start
100 REM now we start the clock
110 LET t1= FN t()
120 LET a=t1/30* PI: REM a is t
    he angle of the second hand
    in radians
130 LET sx=72* SIN a: LET sy=72
    * COS a
140 PLOT 131,91: DRAW OVER 1;sx
    ,sy: REM draw hand
200 LET t= FN t()
210 IF t<=t1 THEN GO TO 200: RE
    M will wait until time for
    next hand
220 PLOT 131,91: DRAW OVER 1;sx
    ,sy: REM rub out old hand
230 LET t1=t: GO TO 120
```

The internal clock that this method uses should be accurate to about 0.01% (approx 10 seconds per day) so long as the ***+3*** is simply running the program. However, when you use the **BEEP** statement (described in part 19 of this chapter) or operate the disk drive or any peripheral attached to the ***+3*** (eg a printer or second disk drive), the internal clock stops temporarily, losing time.

The numbers PEEK 23674, PEEK 23673 and PEEK 23672 are held inside the +3 and used for counting in 50ths of a second. Each is between 0 and 255 and they gradually increase through all the numbers from 0 to 255, after 255 they drop straight back to 0.

The one that increases most often is PEEK 23672 - every 1/50 second it increases by 1. When it is at 255, the next increase 'nudges' it to 0, and at the same time it increments PEEK 23673 up by 1. When (every 256/50 seconds) PEEK 23673 is nudged from 255 to 0, it in turn increments PEEK 23674 up by 1. This should be enough to explain why the expression above works.

Now, consider this carefully. suppose our three numbers are 0 (for PEEK 23674), 255 (for PEEK 23673) and 255 (for PEEK 23672). This means that it is about 21 minutes after switch on. Our expression ought to yield (65536x0+256x255+255)/50 which is equal to 1310.7.

But there is a hidden danger - the next time there is a 1/50 second count, the three numbers will change to 1, 0 and 0. Every so often this will happen when you are half way through evaluating the expression - the +3 would evaluate PEEK 23674 as 0, but then change the other two to 0 before it can PEEK them. The answer would then be (65536x0+256x0+0)/50 which is equal to 0, which is obviously wrong.

A simple way of avoiding this problem is to evaluate the expression twice in succession and take the larger answer.

Now if you look carefully at the previous program, you can see that it does this implicitly.

Here is a trick to apply the rule. Define the functions.

```
10 DEF FN m(x,y)=(x+y+ ABS (x-
   y))/2: REM the larger of x
   and y
20 DEF FN u()=(65536* PEEK 236
   74+256* PEEK 23673+ PEEK 23
   672)/50: REM time (may be w
   rong)
30 DEF FN t()= FN m( FN u(), F
   N u()): REM time (correct)
```

You can change the three counter numbers so that they give the real time instead of the time since the +3 was switched on. For instance, to set the time at 10.00am you work out that this is 10 x 60 x 60 x 50 which is equal to 1800000 fiftieths of a second (and 1800000 is equal to 65536 x 27 + 256 x 119 + 64 x 1).

To set the three numbers to 27, 119 and 64 you type

```
POKE 23674,27: POKE 23673,119: POKE 23672,64
```

(In countries with mains frequencies of 60 Hz (cycles per second), these programs must replace 50 by 60 where appropriate.)

The function **INKEY$** (which has no argument) reads the keyboard. If you are pressing just one key, (or say, **CAPS SHIFT** and just one other key), then the result is the character which that key gives normally, otherwise the result is an empty string.

Try this program, which works like a typewriter:

```
10 IF INKEY$ <> "" THEN GO TO
   10
20 IF INKEY$ ="" THEN GO TO 20
30 PRINT INKEY$ ;
40 GO TO 10
```

Here line 10 waits for you to lift your finger off the keyboard, and line 20 waits for you to press a new key.

Unlike **INPUT**, **INKEY$** doesn't wait for you, so you don't have to press **ENTER**.

Exercises.

1. What happens if you miss out line 10 in the typewriter program?

2. Another way of using **INKEY$** is in conjunction with **PAUSE**, as in this alternative typewriter program:

```
10 PAUSE 0
20 PRINT INKEY$ ;
30 GO TO 10
```

To make this work, why is it essential that a pause should not finish if it finds you already pressing a key when it starts?

3. Adapt the [illegible] PLAY [illegible]

Part 19
Sound

Subjects covered...

BEEP, PLAY

As you will have already noticed, the *+3* can make a variety of noises. To get the best quality of sound, it's important to make sure that your TV is tuned-in properly (see chapter 2). If, instead of a TV, you are using a VDU monitor (which won't reproduce the *+3*'s sound), note that a separate sound signal (which may be connected to an audio amplifier powering speaker(s) or headphones) is available from the **TAPE/SOUND** socket at the back the *+3*. Headphones may not be plugged into the **TAPE/SOUND** socket directly.

Connections to the **TAPE/SOUND** socket are described in chapter 10.

To get the most out of the *+3*'s musical ability, it helps to have a little knowledge about musical terms.

Note that in the examples that follow, it is important that you type in the string expressions *exactly* as shown in upper case and lower case letters, ie. the example `"ga"` should *not* be typed in as `"Ga"`, `"gA"` or `"GA"`.

Type in this command (don't worry about what it means just yet):

```
PLAY "ga"
```

Two notes were played - the second slightly higher than the first. The difference between the notes is called a *tone*. Now try:

```
PLAY "g$a"
```

Again there were two notes played - the first one was the same as the previous example, but there was less of a jump to the second. If you didn't hear the difference, then try the first example followed by the second again. The second example has half the difference between notes, and this is called a *semitone*.

When you're happy with semitones, try this:

```
PLAY "gD"
```

This sort of difference is called a *fifth*, and occurs quite often in music of all kinds. With that example ringing in your ears, type:

```
PLAY "gG"
```

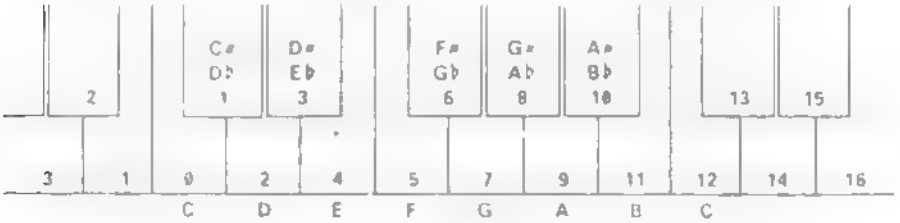
Although (hopefully) you noticed that there was a much bigger difference that time than for the fifth, the two notes somehow sounded much more similar. This is called an ***octave*** and is the point at which music starts to repeat itself. Don't worry about that unduly, just remember what an octave sounds like.

There are two ways of making music and sounds with the ***+3***. The most elementary is the somewhat spartan **BEEP** command. This takes the form

BEEP duration, pitch

where the duration and pitch parameters represent numerical expressions. The duration is given in seconds, and the pitch is given in semitones above middle C (negative numbers for notes below middle C).

Here is a diagram to show the pitch values of all the notes in one octave on the piano for **BEEP**.



Hence, to play the A above middle C for half a second, you would use

```
BEEP 0.5,9
```

and to play a scale (for example, C major) a complete (albeit short) program is needed.

```
10 FOR f=1 TO 8
20 READ note
30 BEEP 0.5,note
40 NEXT f
50 DATA 0,2,4,5,7,9,11,12
```

To get higher or lower notes, you have to add or subtract 12 for each octave that you go up or down.

BEEP exists mostly to provide compatibility with the older designs of Spectrum, though it can be useful for very short or rapid sounds in your own programs you develop. The second way of [illegible] is achieved using the **PLAY** command (if you [illegible] in this section, you'll remember that that's what you [illegible]).

PLAY is much more flexible than BEEP - it can play up to three voices in harmony with all manner of effects, and gives a much higher quality of sound. It's also much easier to use. For example, to play A above middle C for half a second, simply type in:

```
PLAY "a"
```

and to play the C major scale (which needed a program to itself before), use

```
PLAY "cdefgabC"
```

Notice that the last C in the example above is in upper case. This tells the PLAY command to play it an octave higher than the lower case c. A *scale* by the way is the term used for a set of notes spanning an octave. The example above is called the C major scale because it's the set of notes between two C's. Why major? There are two main classes of scale: major and minor, and this is just musical shorthand for describing two different sets. Just for interest, the C minor scale sounds like this:

```
PLAY "cd$efg$a$bC"
```

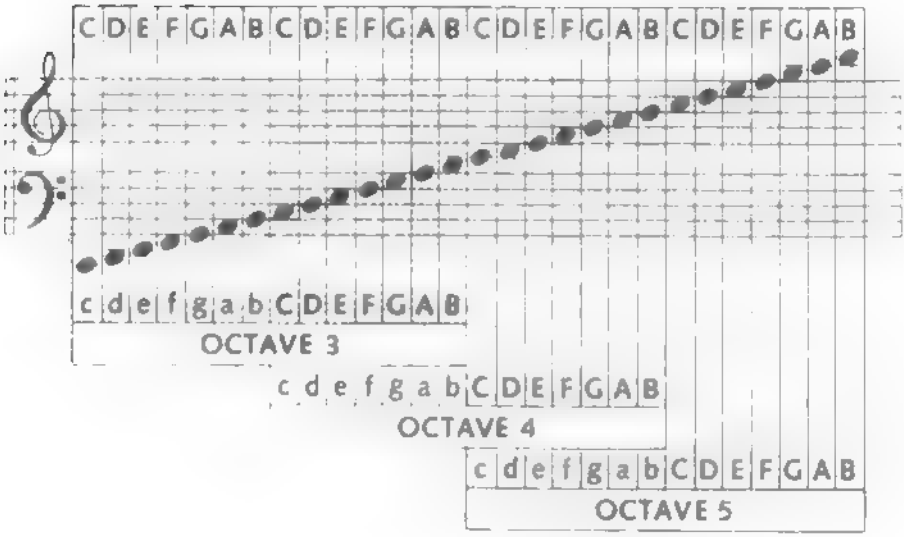
Preceding a note by $ drops it by a semitone (*flattens* it), and preceding a note by # raises it by a semitone (*sharpens* it). The PLAY command spans 8 octaves, and can be told which one to use by having the upper case letter O followed by a number in the list of notes it is given. Type in this little program:

```
10 LET o$="O5"
20 LET n$="DECcg"
30 LET a$=o$+n$
40 PLAY a$
```

There are a few new things in this program. Firstly PLAY is just as happy with a string variable as with a string constant. In other words, providing that a$ has been set up beforehand PLAY a$ works just as well as PLAY "O5DECcg". In fact, using variables in PLAY statements has certain distinct advantages, and we shall be doing this from now on.

Notice also that the string a$ has been built up by combining the two smaller strings o$ and n$. While this doesn't make much difference at this sort of level, PLAY can cope with strings many thousands of notes long, and the only sensible way of creating and editing those strings from BASIC is to combine lots of smaller strings in this way.

Now run the above program. Edit line 10 so that "O5" becomes "O7", and run it again, or if you want to be a big spaceship make it "O2". If you don't specify an octave number for a particular string, then the +3 assumes that you want octave 1. Here follows a diagram of the notes and octave numbers which correspond to the standard even-tempered musical scale:



There is a lot of overlap so for example "03D" is the same as "04d". This makes it easier to write tunes without having to change octave all the time. Some of the notes in the lowest octaves (0 and 1) aren't very accurate for technical reasons and so the computer just makes a brave attempt at getting as close as possible.

PLAY can also handle many different lengths of note. Edit the program above so that line 10 is now

```
10 LET o$="2"
```

and run it. Then alter the setting of o$ between "1" and "9". The note length can be changed anywhere in a string by including a number between 1 and 9 and this is effective for all subsequent notes until a new number is encountered. Each of these nine note lengths has a specific musical name and looks different when written down in musical notation. The following table shows which is which.

NUMBER	NOTE NAME	MUSICAL SYMBOL
1	semi-quaver	𝅘𝅥𝅯
2	dotted semi-quaver	𝅘𝅥𝅯.
3	quaver	♪
4	dotted quaver	♪.
5	crotchet	♩
6	dotted crotchet	♩.
7	minim	𝅗𝅥
8	dotted minim	𝅗𝅥.
9	semi-breve	𝅝

PLAY [illegible] *triplets* [illegible]

NUMBER	NOTE NAME	MUSICAL SYMBOL
10	triplet semi-quaver	♫
11	triplet quaver	♫
12	triplet crotchet	♩♩♩

PLAY [illegible] "&" [illegible]

```
10 LET o$="04"
20 LET n$="DEC&cg"
```

[illegible] PLAY [illegible] "5_7c" [illegible]

[illegible]

```
10 LET o$="062"
```

seems a good bet - but no. The computer will find the 0 and try to read the number following it. When it finds 62 it will stop with the report n Out of range. In cases like this there is a dummy note called N that just serves to split things up, so line 10 should be

```
10 LET o$="06N2"
```

The volume can be set between 0 (minimum) and 15 (maximum) using "V" followed by a number. In practice only 10 to [illegible] are likely to be useful, as 1 to 9 are too soft unless the *+3* is being used with an amplifier. As previously mentioned BEEP is louder than a single channel of PLAY, but if all three channels play a note at volume [illegible] then it should be at the same level as a note produced by BEEP.

Playing more than one channel at a time is very simple, you just separate lists of notes by commas. Try this new program:

```
10 LET a$="04cCcCgGgG"
20 LET b$="06CaCe$bd$bD"
30 PLAY a$,b$
```

In general there is no difference between the three channels, and any string of notes can be put onto any channel. The overall speed of the music (the tempo) must be in the string assigned to channel A (the first string after PLAY), otherwise it will be ignored. To set tempo in beats (crotchets) per minute use "T" followed by a number between [illegible] and 240. The standard value is 120, or two crotchets per second. Modify the program above to:

```
 5 LET t$="T120"
10 LET a$=t$+"04cCcCgGgG"
20 LET b$="06CaCe$bd$bD"
30 PLAY a$,b$
```

and run it several times, changing line 5 for different tempos.

A common feature of music is the repetition of a group of notes. Any part of a string can be repeated by enclosing it in brackets. Now if you change line 10 to:

```
10 LET a$=t$+"04(cC)(gG)"
```

PLAY treats it just the same as the old line. If you leave a closing bracket (with no matching opening bracket) then the string up to that point is repeated indefinitely. This is useful for rhythm [illegible] demonstrate, try this (you will have to press **BREAK** to stop the sound):

```
PLAY "04N2cdefgfed)"
```

or:

```
PLAY "04N2cd(efgf)ed)"
```

If you set up an infinitely repeating bass line, and then play a melody with it, then it would be nice if the bass line stops when the melody does. There is a device to do this - if **PLAY** comes across **"H"** (for halt) in any of the strings it is playing, then it stops all sound immediately. Run the following program (again, you'll have to press **BREAK** to stop it)

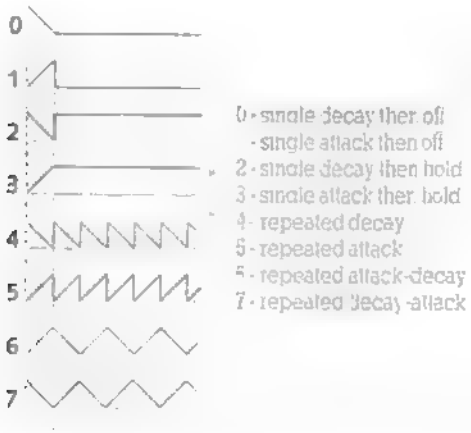
```
10 LET a$="cegbdfaC"
20 LET b$="04cC)"
30 PLAY a$,b$
```

Now modify line 10 to:

```
10 LET a$="cegbdfaCH"
```

and run it again.

So far we've only used notes which start and stop at one level of volume. The ***+3*** can alter the volume of a note while it is playing, so it can start loud and die away like a piano, or rise and fall like a dog growling. To turn these effects on, use **"W"** (for waveform) followed by a number between 0 and 7, together with **"U"** for each channel you want to use the effect on. Any channel with a volume setting (**"V"**) will not respond to **"U"**. This table shows graphically how the volume changes for each setting.



This program plays the same note with each effect in turn, so you can compare them against the diagram above.

```
10 LET a$="UX1000W0C&W1C&W2C&W
   3C&W4C&W5C&W6C&W7C"
20 PLAY a$
```

The U turns on effects, and the W selects which waveform to use. There's also an "X1000" X sets how long the effect will last for (from 0 to 65535). If you don't include an X then the +3 will choose the longest value. Waveforms that settle down (0 to 3 in the previous table) after the initial part, work best with X settings of about 1000, whereas repetitive effects (4-7) are more effective with short values like 300. Try varying the X setting in the previous program to get some idea of how each works.

The PLAY command isn't limited to pure musical notes. There are also three 'white noise' generators (white noise is a sound which is like an un-tuned FM radio or TV), and any of the three channels can play notes, white noise, or a mixture of both. To select a mix of noise and note, you may use "M" followed by a number between 1 and 63. You can work out which number to use from this table:

	[illegible]			[illegible]		
	A	B	C	A	B	C
[illegible]		[illegible]	[illegible]	[illegible]	[illegible]	

Write down the numbers corresponding to the effects you want, and then add them together. If you wanted A to be noise, B to be tone, and C to be both tone and noise, then add 8, 2, 4 and 32 to get 46 (the order of the channels is the order of the strings which follow the PLAY command). The best effects can be obtained with the A channel - don't be afraid to experiment.

By now, you'll be writing symphonies! However, it can be difficult to work out just which part of the music a particular section of string is responsible for. To alleviate this problem, your music string may include comments enclosed between ! exclamation marks, for example:

```
1090 LET z$="CDcE3Ge4 6f! end of
     75th bar !egeA"
```

The PLAY command will simply 'hop' over any comments in the string.

If [illegible] trument with MIDI, then the +3 can control it using PLAY. Up to [illegible] synthesisers, drum machines or sequencers. The PLAY command [illegible] in this section, except that each string should include a "Y" [illegible] number between [illegible] and 16. The [illegible] after the Y controls which channel the [illegible] to. Up to eight strings [illegible] be used, the first three strings [illegible] be played [illegible] problem [illegible] the TV sound down [illegible] also [illegible] and MIDI programming [illegible] PLAY [illegible] "Z" [illegible] number. Key velocities (loud [illegible] the V [illegible] "V6" [illegible] and 48 as a key velocity).

So, to send a little tune (in four-part harmony) to a music [illegible] synthesiser (after consulting your synth handbook to find out how it allocates MIDI channels to different voices) you would use the PLAY command with four strings, each starting with Y followed by a number. This example program illustrates the PLAY command in some of its glory:

```
10 LET a$="Y1T10002(((1CCg$b))
   (($E$E$b$D))((FFC$E))((GGDF
   ))))"
20 LET b$="Y205N&&&&C$bfG)"
30 LET c$="Y304((3C&)C&1CCDD(3
   $E&)$E&1$E$EEE(3F&)F&1FF$G$
   G(3G&)G&1GG$EC))"
40 LET d$="Y4N9&&&&&&&&(9EGF7b
   5CD))"
50 PLAY a$,b$,c$,d$
```

Summary table

Finally, here [illegible] list of the parameters that can be used in the string [illegible] PLAY command together with the [illegible] their meanings.

STRING	FUNCTION
a g A G	[illegible]
$	[illegible]
#	[illegible]
0	[illegible]
1 12	[illegible]
&	[illegible]
N	[illegible]
V	[illegible]
W	[illegible]
U	[illegible]
X	[illegible]
T	[illegible]
( )	[illegible]
! !	[illegible]
H	[illegible] PLAY [illegible]
M	[illegible]
Y	[illegible]
Z	[illegible]

Part 20
File operations

Subjects covered...

- Drives
- FORMAT
- Filenames
- SAVE LOAD
- Disk catalog: CAT
- Wildcards
- MERGE
- Deleting and renaming files
- File attributes
- ERASE, MOVE, COPY
- The RAMdisk
- Tape operations
- VERIFY
- Tape catalog: CAT

Drives

The *+3* has a built in disk drive that can be used to save and load your own programs, and to load programs produced by other people. As it is also possible to connect a second disk drive to the *+3* BASIC must have some way of identifying which disk drive is which. The built-in drive is known as drive A (always followed by a colon because *+3* BASIC knows that when you say A you mean 'disk drive A'). If the external drive is present, it is referred to as drive B.

As the processor at the heart of the *+3* can only [illegible] use with 64K of memory at a time, the extra RAM in the *+3* s 128K [illegible] is used just like another drive. This is called the *RAMdisk* and is identified by the letter M (for memory drive). All the commands (except FORMAT) that you can use on drives A and B can also be used on drive M. Drive M is much faster than the mechanical disk [illegible] nber that just like the contents of the program memory, the the [illegible] **RESET** button or switch off the *+3*. BASIC's NEW command, however, will leave any files stored on drive M intact.

If you don't have a second disk drive connected to your *+3* you can still use the machine as if drive B were present. If you ask the machine to perform an operation on drive B (you'll see how to do this later) a message will appear asking you to

```
Please put the disk for B: into
the drive then press any key
```

, whereupon you should put the disk that you would have used in drive B (if it had existed) into drive A, then press any key (for example **ENTER**). From then on, the machine will treat the built-in disk drive as if it really *were* drive B. When the *+3* next needs to perform some operation on the disk that was *originally* in the drive, it will ask you to,

```
Please put the disk for A: into
the drive then press any key
```

This technique will be particularly useful when using the `COPY` command (described later in this section).

Now that you know which drives are available and what they are called, let's see what they can be used for. Type in the short program (which displays coloured squares) that you first met at the end of part 16, ie

```
10 POKE 22527+ RND *704, RND *
   127
20 GO TO 10
```

This is the program that you are going to save onto disk.

As previously explained (in chapter 6), you cannot simply unwrap a brand new disk and hope to save programs onto it straight away, it must first be made ready to use with the aid of the `FORMAT` command. `FORMAT` will erase anything that was previously on the disk and set it up for *+3* BASIC to use. Be careful, therefore, not to `FORMAT` any disk that has programs on it you might like to keep. To format your new disk, type in the following

```
FORMAT "a:"
```

If you haven't already put your new disk into the drive, don't worry - the *+3* will just come back with the error report `Drive not ready`.

In this case, put your new disk into drive A, and re-type the command.

By the way, while you are using the various disk commands you may occasionally see the report `Drive A: not ready` (possibly followed by `- Retry, Ignore or Cancel ?`). This invariably means that you have forgotten to put a disk into the drive.

Whenever a report appears that ends with `- Retry, Ignore or Cancel ?`, there are three options open to you.

The first is to take action to rectify the problem, for example, if the report was `Drive not ready`, then put a disk into the drive and type R (to retry). The disk system will then try to carry out the same operation again and hopefully this time it will succeed.

If you were half way through copying a large file and an error such as `Missing address mark` appeared, this would usually mean that the disk being read has been damaged in some way. Try R a few times and if the error persists, you indeed have a damaged disk. At this stage, you'll probably want to salvage any un-damaged data, so try typing I (to ignore). Although this tells the *+3* s disk system to ignore the error, there is no guarantee that all your data will be read intact - it's really just a last ditch operation when all else has failed.

Finally, if you have tried to perform an operation where an error occurs, you may realise that there is no point in trying to go on. In this case type **C** (to cancel) which tells the ***+3*** s disk system to abandon the current command. Having typed **C** BASIC will report an error (usually very similar to the text of the previous report)

Back to our attempt to format a disk. If you have made a mistake and the disk you put into drive A has already been formatted, the ***+3*** will spot this and you will receive the report

```
Disk is already formatted,
A to abandon, other key continue
```

This is a safety feature that will allow you to abandon the format before the process gets going, if, by some chance, you inserted the wrong disk. In this case you should type **A** (to abandon) and nothing more will happen. If, however you really *do* intend to reformat the disk and don't mind losing what's on it then press any key apart from **A** (eg. press **ENTER**

After about 30 seconds, the usual **0 OK** report will appear. The disk is now usable and should not need to be formatted again. You can always reformat a disk if you wish to clear the disk of data completely, but remember that it is an irreversible process.

Disks can occasionally become spoilt *corrupted*. This can happen if some dust or dirt comes into contact with the disk surface, if the disk is left too close to a magnetic field (such as that produced by a TV or a telephone), if the disk is ejected while it is being written to, or if the disk is left in the drive when the computer is switched on or off. A corrupted disk will cause errors during **LOAD** or **SAVE** and should be reformatted before any more data is saved onto it

If you want to recover files from a corrupted disk, you can try to copy them individually (using **COPY**) to a known good disk. If one or more files prove uncopiable then you have probably lost them for good. We recommend that you keep at least two copies of important files (on different disks) - one for day-to-day use and one kept in a safe place just in case the unthinkable happens. Making regular copies of valuable data and programs is known as ***backing-up*** and is an essential habit to get into. Backing-up can save you untold misery and tears. You can, of course make back-ups onto tape which may prove cheaper in the long run

Unlike many computers the **FORMAT** command is built-in and can be used like any other BASIC command. It doesn't affect the program you have in the computer so you can now save the little two-line program you typed in a moment ago

Type in the following

```
SAVE "squares"
```

The word **SQUARES** is just a name that you use to label the program you are going to store on disk. To prevent confusion, everything stored on disk must be given a name. These names (called ***filenames***) are a little different from those that you may use when storing programs on tape.

Filenames

The range of characters that you are allowed to use for disk filenames is more limited than for tape filenames. The format of filenames used on +3 disks is the same as that used by an operating system known as *CP/M* (Control Program/Monitor) by Digital Research Inc. The fact that these formats are the same means that you can take a +3 disk and use it on other computers. Data can be transferred in this way between the +3 and a CP/M system, and this is most likely to be useful for people writing machine code programs or moving text from a +3 word processor to a CP/M program. (It is extremely unlikely that programs written in BASIC can be usefully converted from one machine to another using this method.)

Filenames can be as simple as the example above - SQUARES (or even simpler - S for example). However, a full CP/M-type filename can be made up of as many as four parts: *user number*, *drive letter*, *name* and *type*. Each of these parts is called a *field* (eg. the name field or the type field).

You needn't worry about what user number means if you don't know already, then it's probably best to remain blissfully ignorant. However, for anyone who is interested, on CP/M machines with very large disk capacities (or hard disks) with perhaps more than one terminal connected, user numbers are used to partition files into subsections (known as user areas) so that there isn't just one huge directory with several thousand files. On the +3 however, disks cannot have more than 64 files, so the use of user areas is not really necessary. Nevertheless, user areas can be specified in filenames used in +3 disk commands. They take the form:

user number drive letter : filename

where user number is in the range 0 to 15, and drive letter is A, B, or M. If you specify a user number, then you must also specify a drive letter. So, to save our example program in user area 5, we would use:

```
SAVE "5a:squares"
```

The problem with using user areas is that it's quite easy to forget which user area you saved a file to, and so finding it could take a while (as the CAT command can only catalog one user area at a time).

As just mentioned, the drive letter will normally be A, B, or M. Notice that the letter must be followed by a colon (eg. a:squares). If you don't specify a drive letter, then +3 BASIC will use the drive that was last used - this is known as the *default* drive. (When you first switch on the +3, the default drive is set to A.) So typing:

```
SAVE "squares"
```

is just as if you had typed:

```
SAVE "a:squares"
```

There are special forms of **SAVE** and **LOAD** which can be used to change the default drive. When **SAVE** or **LOAD** is followed by a filename that contains nothing but a drive letter and a colon, the drive identified by the letter is then made the new default drive. So

```
SAVE "m:"
SAVE "squares"
```

will change the default drive to drive M, then save the program onto drive M. (If the above command had been **SAVE "m:squares"** the program would still have been saved to drive M, though the default drive would not have been changed.)

To switch the default back to drive A, type

```
SAVE "a:"
```

Note that **SAVE** and **LOAD** followed by just a drive letter (and colon) will do nothing other than change the default drive. They certainly won't save or load a program. You must use **SAVE** or **LOAD** followed by a real filename for this.

The name field of a filename is the only field that you have to specify when using **SAVE**, **LOAD** etc. The name field can be from 1 to 8 characters long and may contain any of the following:

Letters	**abcdefghijklmnopqrstuvwxyz** (upper or lower case)
Digits	**0123456789**
Other characters	**"#$'@'_{}~'**

Upper and lower case letters are the same in filenames, so **EXAMPLE** and **example** would be identical.

A filename can end with an optional type field (which is just a further three characters) that you may wish to use in order to group together files of the same type. If a type field is specified, it *must* be preceded by a dot. (Unlike some other BASICs, *+3* BASIC does not automatically allocate a type field to files if one is not specified.) You may find it useful to add your own type fields; a popular convention is to use the type fields **.BAS** to identify BASIC files and **.BIN** to identify **CODE** files (**BIN** being short for binary). If you think this is a good idea, then the previous example program could be saved using:

```
SAVE "squares.bas"
```

The characters * and ? have a special meaning in *+3* BASIC, and cannot be used in a filename for **LOAD** and **SAVE**. There are, however, some commands in which * and ? can be used, and these will be discussed later.

Here are some examples of valid filenames

```
z
squares
m:picture.bin
a:fred
13a:hello
ØM:CAPITALS
test.bas
philip
glass.mus
a:a.a
```

Here are some illegal filenames (and reasons why)

`pac man`	(must not contain any spaces)
`(test)`	(must not contain brackets)
`/<>-+=!&`	(must not contain any of these characters)
`excessive`	(more than 6 characters long)
`.bas`	(has no name field)
`later...`	(only one dot allowed)
`7:dubious`	(if user number is specified then filename must also contain drive letter -eg `7a:dubious`)

Disk catalog

You may have spotted the fact that we have now saved the same program twice with two different names (`SQUARES` and `SQUARES.BAS`). It would be nice to be able to check what has been saved on a particular disk and this is where the `CAT` command comes in. `CAT` displays a catalog of what you have stored on a disk.

Press **ENTER** then type in

```
CAT
```

The **+3** will take a quick look at the disk (the read/write indicator lamp will come on briefly) and will display a list of the disk's contents on the screen. The list is sorted into alphanumerical order and each file is followed by an indication of its size to the nearest number of kilobytes (rounded up). At the end of the list the amount of free space on the disk is also displayed.

`CAT` (on its own) is the simplest form of the command. If you wanted to list all the files on a different drive (eg drive M:) you would use

```
CAT "m:"
```

When **CAT** is followed by a filename containing just a drive letter, ie **A: B:** or **M:** (including the colon), all the files on the nominated drive will be listed. **CAT** on its own gave a list of the files on drive A: this is because A: is the current default drive. You will remember that **LOAD** or **SAVE** followed by a drive letter will make that drive the current default. So..

```
LOAD "m:"
CAT
```

will also list all the files on drive M. This isn't quite the same as **CAT "m:"**, because now the default drive has been left as M. Change it back to A: before going further.

We will now save several copies of our simple example program using different names, so that you will be able to see what the various forms of the **CAT** command will produce. So far the disk should contain **SQUARES** and **SQUARES.BAS**. If either of these weren't listed with the above **CAT** command, add their name to the list below. Type the following.

```
SAVE "fred"
SAVE "fat"
SAVE "santa.bin"
SAVE "trepur.bak"
SAVE "cliff.cjl"
SAVE "sausages.bas"
```

Don't worry about cluttering up the disk with lots of copies of the same thing - you'll be shown how files can be erased later.

Wildcards

If a disk has a large number of files it is often desirable to selectively list only those of interest. The ***+3*** caters for this. If, for example, you wished to list only those files that ended in **.BAS** you would use:

```
CAT "*.bas"
```

The asterisk character ***** is what's known as a ***wildcard***. When a filename, perhaps containing a wildcard, is specified, **CAT** will list only the files that match the specification given. When the ***** wildcard is used (in either the name field or the type field) it means 'any character from here to the end of this field'. So [illegible] and we want **CAT** to display any files that have any characters in their name field and [illegible] **BAS** in their type field. If there are no files on the disk that match the specification, the report **No files found** will be displayed (followed by the amount of free space). If you give a file specification of ***.*** (ie **CAT "*.*"**) or no specification at all (ie **CAT**) and the report **No files found** is displayed, then this means that the disk is empty. An empty disk in drive A: or B: will have a free space value of 173K (this value may be different for disks from other types of computer). An empty drive M: will usually have 58K free. If you catalog a disk containing a commercial program (such as a game) it might appear to have no files on it but very little space free. This is a protection measure taken by the software writers to prevent illicit copying and shouldn't cause any concern.

If we use

```
CAT "s*.bas"
```

then all files that begin with the letter S and then have any characters whatsoever between the S and the end of the name field, followed by a type field of BAS will be shown. Files with the names SQUARES.BAS SAUSAGE.BAS SUPER.BAS would all be listed, but SQUARES.BIN TOAST.BAS and SQUARES would not.

The * wildcard can also be used in the type field of a filename (note, however, that you cannot use it in place of the user number or drive letter). If we wanted to list all files that had SQUARES as their name field and anything as their type field, we would use

```
CAT "squares.*"
```

Similarly, if we wanted to list all the files that began with a letter S and had a type field that began with the letter B we would use

```
CAT "s*.b*"
```

When we type CAT (on its own), we want to list *all* files on a disk. Therefore CAT is just a shorthand way of saying

```
CAT "*.*"
```

From the above you will notice that the * wildcard can only be used as the last character in a field, and it is used to mean 'I don't care what other characters are present between here and the end of this field'. Sometimes, however, you may want to specify a group of files but need to be a little more discerning. This is when you use the ? question mark wildcard (in either the name field or the type field). The ? wildcard means 'I don't mind what character happens to be in this specific position'.

Therefore, if we used the command

```
CAT "?at"
```

then the files listed would be all those that are three characters long, ending in AT, but we don't mind what character is in the first position. Thus files such as CAT SAT MAT and FAT would be listed, but CAR, CATTLE and AT would not. Unlike the * wildcard, ? wildcards can be used in place of any of the 8 characters of the name field and the 3 characters of the type field. There is no limit to the number of question marks you can use (other than the 8 and 3 limits of the filename field lengths).

Valid file specifications containing ? wildcards include:

?.bas	(one letter name with a type field of BAS)
s?uares.*	(specific files whose second character doesn't matter, with any type field)
ca??.?t?	(files beginning CA with four characters in the name field, and with a type field of 3 characters whose second letter is T)
????????.???	(exactly the same as *.*)

If you have a printer connected to your *+3* you may find it useful to print-out the files listed by CAT. You can do this by directing the output from CAT to stream 3 (streams are explained in part 22 of this chapter). The command to do this is:

```
CAT #3
```

If you only want some of the files printed-out, you can also include a file specification in exactly the same way as before. For example:

```
CAT #3,"a:*.bas"
```

(The above CAT #3 commands will not work unless a printer is connected to the *+3* and is on line. To abandon, press **BREAK**.)

Any form of the CAT command may also end with the word EXP, for example, CAT "a:" EXP. The EXP is short for expanded, and as the name might suggest, gives you a little more information about the *attributes* of the files on a disk. Not only will the expanded catalog display system files but also it will indicated whether files are set to write protected mode, archive mode or system status (these terms are explained in the section ahead entitled 'File attributes').

(There is one other specialist use for the CAT command, and this will be dealt with in the section ahead entitled 'Tape catalog'.)

Now that you have successfully saved a program to drive A, you can happily switch off or reset the *+3* or start a NEW program, knowing that you could always load in the saved program if you needed it. Remember - there is a difference between resetting the *+3* and using the NEW command - if you reset, all the *+3*'s memory (RAM) will be cleared. This includes any files you may have saved on drive M. When you use NEW however, any files on drive M will remain intact. As we have saved the program on the disk in drive A, you can go ahead and press the **RESET** button, then release it. The usual opening menu will be displayed. Select +3 BASIC then type the command:

```
LOAD "squares"
```

The LOAD command reads in a new program (and variables) from disk, and then deletes any program (and variables) previously in the memory. (If the program that you specified to load is not in the disk, then any program currently in the memory is not deleted.) Just like SAVE, the LOAD command must be followed by a filename whose name field is at least one character long. If you have been used to a tape machine in the past, then you may have used LOAD "" to mean load the next program on the tape. The concept of a next program on disk does not exist, so if you don't specify a filename, the disk system won't know what to load and will report an error. If you can't remember what name you saved a file under, use CAT to check what's on the disk (this is why it's a good idea to save programs on disk using mnemonic names (names that remind you what they contain) - eg. it is more obvious what sort of program a file named TENNIS.BAS contains compared to one simply named T).

There is a short cut for loading programs (such as games) that have been specially set up - you can select the Loader option from the opening menu. This option, when selected, attempts to load and run programs. First of all, it looks for a program called * on the disk. If this exists, then it will be loaded and run. The program has to be a machine code program saved in a particular fashion (as BASIC can't use * as a filename for SAVE) and is therefore only for use on commercial software or by those who understand machine code.

If * can't be found, the *+3* will then look for a file called DISK. This *can* be a BASIC program that you've previously written and saved, so if the Loader option finds a program called DISK, it will load it and wait for the next operation.

At this point, pressing **ENTER** will just load the program again.

If you wish to run or edit the program after it has loaded, first press the cursor down key ↓ once, then **ENTER**. This selects the +3 BASIC option from the opening menu.

If there isn't a program called DISK on the disk (or if the *+3* detects that there isn't a disk in the disk drive), then the computer will try to load a program from tape, displaying the message

```
Insert tape and press PLAY
To cancel - press BREAK twice
```

This is the recommended method for loading Spectrum *+3* (Spectrum *+2* and Spectrum 128) software from tape (see chapter 4).

As previously mentioned, LOAD deletes the old program and variables in the *+3* whenever it loads in the new ones from disk. However, there is another command - MERGE, which is similar to LOAD but it only deletes an old program line or variable if there is a new one with the same line number or name. Clear the program memory using the NEW command, then type in the dice program from part 1 of this chapter and SAVE it onto disk, using

```
SAVE "dice"
```

Use NEW to clear the program memory again, then enter and run the following program:

```
 1 PRINT 1
 2 PRINT 2
10 PRINT 10
20 LET x=20
```

Now type in:

```
MERGE "dice"
```

If you then LIST the program, you will see that lines 1 and 2 have survived, but lines 10 and 20 have been overwritten by those from the dice program. Note that the value of the variable x has also survived (try PRINT x).

You have now seen simple forms of five of the commands that work in conjunction with disk:

FORMAT	Prepares brand new disks so that programs can be saved onto them. FORMAT can be used to completely erase everything on a disk that has already been used.
SAVE	Stores the program and variables onto a disk.
LOAD	Clears the computer of all its program and variables, and replaces them with new ones read in from disk.

MERGE	Similar to LOAD except that it does *not* clear the old program lines and variables unless it has to (because they are the same as those being loaded in from disk)
CAT	Displays a list of the files contained on a disk

A variant on SAVE takes the form

```
SAVE filename LINE number
```

A program which is saved using this command is stored in such a way that when it is loaded, it automatically jumps to the line with the given number then runs itself

Use NEW to clear the program memory then type in the following

```
10 PRINT "program running"
20 PLAY "cdefgabC"
```

Now save this program using the command

```
SAVE "disk" LINE 10
```

Now reset the +3 and when the main menu appears ensure that the disk (with the above program on it) is in the drive then press the **ENTER** key This will select the Loader option which searches for a file on the disk called DISK When it finds the simple example program you just saved, it will load it and as it was saved using a LINE parameter it will automatically start running (from line 10)

At this point pressing **ENTER** will load and run the program again

If you wish to quit the program after it has run press the cursor down key ⇩ once then **ENTER** This selects the +3 BASIC option from the opening menu

Note that if you load a program called DISK which doesn't automatically run (using the Loader option from the opening menu) then you will have to select the +3 BASIC option (after the program has loaded) before you can run it or edit it

So far the only kinds of information we have stored on disk have been programs (together with their variables) There are also two other kinds of information called *arrays* and *bytes*

You can save arrays on disk using the keyword DATA in a SAVE statement

```
SAVE filename DATA array name ( )
```

where filename is the name that the information will have on disk and works in exactly the same way as when you save a program

The array name specifies the array you want to save so it is just another (p a le ter followed by $) Remember to put the brackets ( ) after the array [illegible]

Be clear about the separate roles of filename and array name. If you say (for instance)

```
SAVE "bloggs" DATA b ()
```

then **SAVE** takes the array **b()** from the computer and stores it on disk under the name **BLOGGS**.

The command

```
LOAD "bloggs" DATA b ()
```

sees if it is possible to load the array (ie if there is room for it in the computer), then if so, deletes any already existing array called **b()** and loads in the array **BLOGGS** from disk, calling it **b()** in the computer.

You cannot use **MERGE** with saved arrays.

You can save character (string) arrays in exactly the same way. However, note that when you load in a character array, it will delete not only any previous character array with the same name, but also any simple string variable with the same name.

When dealing with a large amount of data you may find it useful to use the **SAVE DATA** option and the **LOAD DATA** option to and from drive M. Once saved on drive M, the space previously used by an array can be re-used. Using drive M will mean that saving and loading are very fast.

Byte storage is used for pieces of information without any reference to what the information is used for - it could be a screen display, or perhaps some user-defined graphics, or just something you have made up for yourself. It is specified using the word **CODE**, as in

```
SAVE "picture.bin" CODE 16384,6912
```

The unit of storage in memory is the byte (a number between 0 and 255) and each byte has an address (which is a number between 0 and 65535). The first number after **CODE** is the address of the first byte to be stored on disk; the second number is the amount of bytes to be stored. In our case, **16384** is the address of the first byte in the file (which contains the screen display), and **6912** is the amount of bytes in it, so we are saving an actual copy of the screen display onto disk. Try the above **SAVE** command. (You don't have to save the bytes using the name **PICTURE.BIN** - it's merely a convenient reminder of what's on the disk.)

To load it back, use

```
LOAD "picture.bin" CODE
```

You can put parameters after **CODE** in the form

LOAD filename **CODE** start,length

Here, the length parameter is used as a safety measure - when the computer attempts to load the bytes from disk, it will check the length and refuse to load the bytes if there are more than specified (thereby safeguarding against the extra bytes accidentally overwriting an area of memory that you wished to preserve). In such a case, the report `Code length error` is displayed. (Anyone using a cassette unit under 48 BASIC should note that the above error will display a different report `R Tape loading error`.)

If you leave out the length parameter, the **+3** will read in the bytes however many there are.

The start parameter shows the address where the first byte is to be loaded back to - this can be different from the address it was saved from, though if they are the same, then you can leave out start parameter in the `LOAD` statement.

`CODE 16384,6912` is such a useful area of memory (the screen display) to save and load, that a special function `SCREEN$` has been provided to represent it, so you can type (for example)

```
SAVE "picture.bin" SCREEN$
```

and

```
LOAD "picture.bin" SCREEN$
```

Automatic back-ups

If you have saved one or two things on a disk and then you save something with a filename that has already been used, what will happen? Well, each time you save a program, the disk system checks to see if the filename you specify has already been used. If it has, the existing copy on disk is given a new filename before the information you have asked to save is stored. The new name given to the existing file has the same name field but its type field will always be `.BAK` (short for backup).

If a `.BAK` version of the file already exists, then that will be lost in preference to the new `.BAK` file. This means that as you save successive versions of a program with the same name the previous copy will still be there in a file called filename`.BAK`. So if you make a serious programming error and inadvertently save the program, you can delete the newest version and rename the `.BAK` file to the original filename. The next section shows you how to do this, but first, type

```
SAVE "a:squares"
```

to save the program using the filename `SQUARES` yet again.

Deleting and renaming files

Files can be deleted from a disk using the **ERASE** command. This should be followed by a filename that specifies which file or files are to be deleted. Just like **CAT** you can use the wildcards * and ? to identify a group of files, or you can specify the name in full if you only want to get rid of one particular file. If you specify a single filename, that file will immediately be erased from the disk - so take care. If you specify a group of files (by including * or ?) BASIC will ask you to confirm that you really mean to delete this group of files. Typing Y will make the deletion process continue, so if you have made a mistake, type N.

If, for example, you wanted to delete a file from drive M called **FRED.BAS** you would use:

```
ERASE "m:fred.bas"
```

If drive M: has already been set as the default drive, then you don't need to include the M: at the start of the filename. It doesn't hurt to include the drive letter anyway, and with as powerful a command as **ERASE** you might feel safer if you do. To erase all the files on drive B, you would use:

```
ERASE "b:*.*"
```

Before doing this, BASIC will ask for confirmation:

```
Erase b:*.* ? (Y/N)
```

and assuming that you really mean to wipe all the files from the disk in drive B, you would then type Y.

If you ask to delete a single file (or a group of files using the wildcards * and ?) and there are no files on the disk that match the specification, then the report **File not found** will be displayed.

Note that **ERASE** followed by just a drive letter (eg. **ERASE "m:"**) will erase *all files* on the specified drive *without asking for confirmation*. Be careful, therefore, not to enter this form of the command unless you really mean to delete everything. (The **ERASE** process will stop and report an error if a write protected disk or file is detected.)

The disk you have been using up to now has many copies of the simple **SQUARES** program (saved under different names) on it. This is a waste of space so you might as well erase those that aren't needed. What you want to do in effect is erase everything except **SQUARES** (though there is no simple way to do this). However, some of the different files have the same [illegible] common, so you may be able to use various forms of * and ? specifications to cut down the amount of typing. See if you can work out the fewest number of **ERASE** commands to erase all files other than **SQUARES**.

Once a file has been saved, it can be [illegible] **MOVE** [illegible] there is [illegible] M: called **SQUARES** [illegible] **BLOCKS** [illegible] **SQUARES** [illegible]

```
SAVE "m:squares"
MOVE "m:squares" TO "m:blocks"
CAT  "m:"
```

Imagine we had saved a file called FRED and then after working on it and saving a new version with the same name, realised that we had made a terrible mistake and would like to recover the last version. This would be possible using the commands

```
ERASE "fred"
MOVE "fred.bak" TO "fred"
```

Unlike ERASE you cannot include the wildcards * or ? when renaming files.

MOVE will take into account the current default drive so the filename doesn't necessarily have to contain a drive letter. Note, however, that it is not possible to use MOVE to rename files between *different* drives. The command

```
MOVE "a:fred" TO "b:eric"
```

(for example) will fail with the error No rename between drives reported. Instead you can use the COPY command (explained ahead) followed by ERASE to achieve the desired result.

File attributes

MOVE has another use besides renaming files. It can also be used to change the *attributes* of a file. Attributes are bits of information associated with a file that tell you (and the computer) a little more about it.

There are three attributes that can be changed. The most useful attribute is write protection. Once a file's write protection attribute has been set, it will not be possible to erase it (or save a file with the same name) until you remove the write protection. It behaves a little like the write protect hole on the disk, but works just on individual files. Unlike the write protect hole, however, it offers no protection against FORMAT, which erases everything on a disk, regardless of attributes. You can set a file's write protection attribute to *on* with a command such as

```
MOVE "squares" TO "+p"
```

The letter P is short for protection (against overwriting). If you now try to use the command..

```
ERASE "squares"
```

you will receive an error report saying File is read only.

To switch write protection *off* use

```
MOVE "squares" TO "-p"
```

and you'll be able to erase the file as before.

In all the MOVE commands that change attributes, + means switch it on, and - means switch it off.

When you are using MOVE to change attributes, the filename can include the wildcards * and ? So to make all the files on drive M write protected, you would use

```
MOVE "m:*.*" TO "+p"
```

As always, the drive letter can be omitted if this is the current default drive.

You can repeatedly switch attributes on or off without causing an error, so if you set write protect on a file that has already got write protection, it will just stay protected.

The second attribute that can be changed is known as the system status attribute. This is really provided just to be compatible with other CP/M based computers, however, if you do set a file's system attribute to on, then you will notice that the file no longer appears in the list of files when you use CAT. The system status attribute is identified by +S (or -S) in the MOVE command. If you use the expanded catalog (ie. CAT EXP), all the files will then be listed including system status files (which are followed by the letters SYS). You may also notice that any files that are write protected are followed by the letters PROT. You can use the system attribute to remove files from a catalog if they would otherwise just clutter things up.

Bear in mind that you cannot have two files on the same disk with the same filename and different system status attributes, so if you try to create or copy a file onto a disk where a file of that name already exists (but is hidden from CAT), then the previous file will be deleted.

The final attribute you can change is known as the archive attribute. In an expanded catalog it shows up as ARC, and is identified by +A (or -A) in the MOVE command. On the *+3* the archive attribute is of no practical use and is only provided for file compatibility with CP/M based computers.

Here are some attribute-setting MOVE commands. See if you can predict what they will do.

```
MOVE "*.*" TO "+p"
MOVE "*.bas" TO "-s"
MOVE "s???.*" TO "+a"
MOVE "m:?.?" TO "-p"
```

If you try to use any letter other than A, S or P in setting or resetting attributes, or if the attribute string is not two characters long, then you will receive the report Invalid attribute.

Quite often, a situation will arise when you would like to make a copy of one of your files (to give to a friend perhaps, or to put it on drive M, so that it can be accessed quicker). The COPY command can be used to copy files from one drive to another and even to make complete copies of disks. The very simplest form of the COPY command will look something like this:

```
COPY "a:fred" TO "m:"
```

This means: put a copy of the contents of the file called FRED (which is presently on drive A:) onto drive M. As no destination name has been specified (after M:) the new file will also be called FRED.

The name before the word TO is known as the *source* filename, and the name after TO is the *destination* filename.

The command

```
COPY "fred" TO "eric"
```

will take the contents of a file called FRED on the default drive and copy it to a file called ERIC also on the default drive. The files FRED and ERIC then contain the same information.

You cannot copy one file to another with the same name and on the same drive. Trying to do so will result in the error report File already exists (or possibly File already in use).

[illegible] * [illegible] ? [illegible]

```
COPY "a:*.ovl" TO "m:"
```

will work (assuming that there are some files on drive A that match this specification) and transfers all files on A with a .OVL type field onto drive M. However, the command

```
COPY "a:*.bas" TO "m:*.bin"
```

will fail with the error report Destination cannot be wild.

The COPY command does not copy any attribute information associated with a file: you have to set any attributes you require on the new file after copying.

COPY will always list the files it is copying in two columns. This will allow you to check that any wildcard specification you use encompasses all the files that you were intending to copy.

After copying, a report will appear to let you know how many files were copied. (If you were copying a group of files, this may be useful to check that you have indeed copied all the files you intended to.)

There is a special form of the COPY command as follows:

```
COPY "a:" TO "b:"
```

which will perform a complete sector by sector copy of the disk in drive A to an already formatted disk in drive B. Anything already stored on the disk in drive B will be lost - so if there are only a few files on the source disk to be copied it may be quicker to use

```
COPY "a:*.*" TO "b:"
```

[illegible] if you have only one drive (A:) you can use the fact that the single mechanism can be used as if it were [illegible] A: or drive B:. For example, suppose you have a single drive system and want to copy a number of files that both end in .BAS from one disk to another. Put the source disk in the drive and type

```
COPY "a:*.bas" TO "b:"
```

Once the *+3* has read part of the first file that ends in **.BAS** it will ask you to

```
Please put in the disk for B: into
the drive then press any key
```

Simply follow this instruction. After the *+3* has written the information onto the drive B disk, it will ask you to..

```
Please put in the disk for A: into
the drive then press any key
```

This process of swapping between disks will go on until all files have been copied. Because the **COPY** command will try to use any free space on drive M, it is a good idea to clear drive M (if possible) before doing a lot of copying (as this can reduce the number of disk swaps needed)

As well as copying files between drives **COPY** can also be used to copy files to the screen or to a printer (if connected). The command

```
COPY "words.txt" TO SCREEN$
```

will display the contents of a file on the default drive called **WORDS.TXT**. Any control characters (except carriage returns) will be filtered out. This command cannot really be used to look at BASIC program files as they contain various control codes. Its main use will be to inspect the contents of ASCII text files such as those produced by a word processor.

The command

```
COPY "words.txt" TO LPRINT
```

is similar to the above, but this time the contents of the file will be sent to the printer. In this case however control codes *will* be sent to the printer. If you have set the print output to be via the RS232 with tokens unexpanded (using **FORMAT LPRINT "R";"U"**) then this command can be used to 'export' files to other computers. Once again this command cannot be used for BASIC programs - it is intended for sending ASCII text files only.

People writing machine code programs may find it easier to do so on a larger development machine. However the files produced by this method will probably not be recognised by the *+3* as BASIC expects to find a 128 byte *header* at the start of each file which contains information used by the **LOAD** command. However once a binary file has been produced on a *+3* formatted disk, it can have a header of the correct type put on it using a command such as

```
COPY "game.com" TO SPECTRUM FORMAT
```

This will produce a new file on the same drive having the same name field but with a type field of **.HED** (short for headed). In the above example a new file called **GAME.HED** will be created and it will be written to the default drive (as no drive letter was specified).

Obviously this command will only be of use for machine code files. Headed files produced in this way will have the length part in their header set to the correct value and the type part set to be a **CODE** file. However, BASIC cannot know what address the file should be loaded to, so the load address should be specified when the **LOAD CODE** command is used. For example, if the above program had been assembled to execute at 7000h (the *h* denotes a hexadecimal number) or 28672 decimal, then the headed file could be loaded with the command

```
LOAD "game.hed" CODE 28672
```

As **SCREEN$** files are just another type of **CODE** file, this technique can be used to 'import' screens designed on another machine, though they obviously wouldn't make much sense unless they had been tailored to fit the ***+3***'s size and layout.

The RAMdisk

You may have been wondering what point there is in storing information in the RAMdisk (drive M:) as it will be lost once the ***+3*** is switched off. Well, perhaps the most obvious use of drive M: is to store chunks of BASIC program (or routines) which can be merged (using **MERGE M:**filename) into a smaller program in sequence. This makes it possible to write about 90K of BASIC program and hold it in the ***+3*** (though to do this, the program structure has to be well defined).

You can keep the various routines on a 3 inch disk and use **COPY** to put them into drive M: before you run the program. The benefit of doing this is that drive M: is much quicker to access than the mechanical drives (A: and B:). The mechanical drives, however, can hold much more data, so you might like to evolve a system using both disk and RAMdisk. Careful design and planning will repay itself many times over in terms of speed and performance.

One of the more interesting uses of the RAMdisk is in animation, where a series of pictures can be defined by a slow BASIC program, stored in drive M:, then called back to the screen at high speed. The following program offers a taste of this. Doubtless you can do better.

```
 10 INK 5: PAPER 0: BORDER 0: C
    LS
 20 FOR f=1 TO 10
 30 CIRCLE f*20,150,f
 40 SAVE "m:ball"+ STR$ (f) COD
    E 16384,2048
 50 CLS
 60 NEXT f
 70 FOR f=1 TO 10
 80 LOAD "m:ball"+ STR$ (f) COD
    E
 90 NEXT f
100 BEEP 0.01, 0.01
110 FOR f=9 TO 2 STEP -1
```

```
120 LOAD "m:ball"+ STR$ (f) COD
    E
130 NEXT f
140 BEEP 0.01, 0.01
150 GO TO 70
```

Before running this program, always make sure that drive M is empty. If it isn't, first type **ERASE "m:*.*"** (typing Y at the (Y/N) prompt), then **RUN**.

Note that in line 40 of this program, the two numbers following **CODE** are the address in memory of the start of the screen (**16384**) and the length of the top third of it (**2048**). By saving and loading only the top third, the overall speed is maintained.

Tape operations

(See chapter 10 (Peripherals for your *+3*) for details on how to connect a cassette unit to your *+3*.)

Much of what has been said in this section about the use of **LOAD**, **SAVE** and **MERGE** on disk will apply equally on tape (if you have connected a cassette unit to the *+3*). However, the commands **FORMAT**, **COPY**, **MOVE**, **CAT** and **ERASE** do not apply on tape (although there is a special form of **CAT** that can be used - described in the section ahead entitled 'Tape catalog').

As you will already know, when you first switch on the *+3* the default drive for all file operations is set to drive A. This means that if you use **CAT**, **ERASE**, **LOAD**, **SAVE**, etc. without specifying a drive letter, then *+3* BASIC will perform the operation on drive A. You will also know that the default disk can be changed using either

```
LOAD "drive letter :"
```

or

```
SAVE "drive letter :"
```

where drive letter is either **A:**, **B:** or **M:** (which must include the colon). In fact you can also use **T:** as a drive letter, but only in this one special form of the **LOAD** and the **SAVE** command...

```
LOAD "t:"
```

After **LOAD "t:"**, all subsequent **LOAD** and **MERGE** operations are performed to tape (until changed back to disk by, for example, **LOAD "a:"**). Similarly, if you use:

```
SAVE "t:"
```

...then all future **SAVE** operations will be performed to tape (again, until changed back to disk by, for example, **SAVE "a:"**). Unlike **A:**, **B:** or **M:**, when you use **T:** as the drive letter, it will change only future **LOAD**, **SAVE** and **MERGE** commands. The default drive used for **MOVE**, **COPY**, **CAT** and **ERASE** will stay the same as it was before (as these commands have no relevance to tape).

If all this sounds a little complicated, a few examples might help to make it a little clearer. Assuming you have just switched on (or reset) the +3, the default for all operations will be A. So if you now type:

```
SAVE "m:"
```

then the default drive for all subsequent operations will be set to drive M (this is exactly the same as if the command `LOAD "m:"` had been used). Using the command:

```
LOAD "b:"
```

will then set the default drive for all operations to drive B. For this sort of thing, **LOAD** and **SAVE** operate in exactly the same way.

If we now use the command:

```
SAVE "t:"
```

this will perform all future **SAVE** operations to tape, but all other commands will still default to drive B. Using the command:

```
LOAD "t:"
```

will also perform all future **LOAD** and **MERGE** operations to tape; however, the default drive for all disk-only commands will still be drive B.

Finally, using the command:

```
SAVE "a:"
```

will perform all future **SAVE** operations and all disk operations (except **LOAD** and **MERGE**) to drive A. **LOAD** and **MERGE** will still be from tape, however.

Let's try to save our simple squares program onto tape. Reset the +3, then type:

```
LOAD "squares"
```

This should load in the program that we saved earlier. If you press **ENTER**, the program will be listed as follows:

```
10 POKE 22527+ RND *704, RND *
   127
20 GO TO 10
```

This is the program that you are now going to save onto tape. Any standard tape should work, although low noise tapes are preferable.

Type in the following

```
SAVE "t:"
SAVE "squares"
```

This will save the program onto tape using the filename **SQUARES** When saving files on tape you are allowed up to ten characters in the name Unlike disk you can use any characters you like and the name can include spaces

The ***+3*** will display the message

```
Press REC & PLAY, then any key.
```

We shall first go through a dry run so that you can see what will happen when we actually do save the program later This time therefore, don't press **REC** and **PLAY** on your cassette unit - just press a key on the ***+3*** (for example **ENTER**) and watch the border around the screen display You will see patterns of coloured horizontal stripes as follows

Five seconds of red and cyan stripes moving slowly upwards followed by a very short burst of blue and yellow stripes

A short pause

Two seconds of the red and cyan stripes again followed by another short burst of blue and yellow stripes

While the stripes appear on the screen you can also hear the sound of the data through your TV's speaker

Keep trying out the above **SAVE** command (without actually operating your cassette unit) until you can recognise these patterns What's actually happening is that the information is being saved in two blocks and both blocks have a 'lead-in' (which corresponds to the red and cyan stripes) followed by the information itself (which corresponds to the blue and yellow stripes) The first block is a preliminary one containing the name and various other bits of information about the program, and the second is the program itself together with any variables present The pause between them is just a gap

Now let's actually save the program onto tape

1. Wind the tape to an area that is either blank or that you are prepared to overwrite

2. Type

```
SAVE "squares"
```

3. Follow the instructions on the screen, ie

```
Press REC & PLAY, then any key.
```

4. Watch the screen as before. When the +3 has finished (with the report 0 OK), stop the tape.

Whenever you save a program to tape before clearing the saved program from the +3's memory, you should always make sure that the program was correctly saved. You can check the signal on the tape against the program in the memory using the VERIFY command (this command isn't used on disk, as disks are not prone to the same sorts of errors as tapes are).

1. Rewind the tape to just before the point at which you saved the program.

2. Type:

```
VERIFY "squares"
```

Play the tape. The border will alternate between red and cyan until the +3 finds the program that you specified, then you will see the same pattern as you did when you saved the program. During the pause between the blocks the message Program: squares will be displayed on the screen. (When the +3 is searching for something on tape, it displays the name of everything it comes across.) If, after the pattern has appeared, the +3 displays the report 0 OK, then your program is safely stored on tape and you can skip to the section ahead entitled 'Verified OK'. Otherwise, something has gone wrong - take the following steps to find out what.

If the program name has not been displayed, then either the program was not saved properly in the first place, or it was but 'was not read back' properly. You need to find out which. To see if it was saved properly, rewind the tape to just before the point at which you saved the program, then play it back while listening to the TV's speaker. The (red and cyan) lead-in should produce a clear, steady high pitched note, while the (blue and yellow) information part gives a much harsher screech.

If you do not hear these noises, then the program was probably not saved. Check that you were not trying to save the program onto the plastic leader at the beginning of the tape. When you have checked this, try saving again.

If you can hear the sounds as described, then SAVE was probably alright and your problem is with reading back.

It could be that you mistyped the program name when you saved it (in which case when the +3 finds the program it will display the mistyped name on the screen). On the other hand, perhaps you mistyped the program name when you verified it, in which case the +3 will ignore the correctly saved program, and carry on 'looking for the wrong name' flashing red and cyan as it goes.

If there is a genuine mistake on the tape, then the +3 will display the report R Tape loading error which means in this case that it failed to verify the program. Note that a slight fault on the tape itself (which might be almost inaudible with music) can wreak havoc with a computer program. Try saving the program again, perhaps on a different part of the tape (or a different tape altogether).

Verified OK

Now let us suppose that you have saved the program and successfully verified it. Loading it back into the memory is just a matter of typing

```
LOAD "squares"
```

(Since the program verified properly, you should have no problem loading it.)

LOAD deletes the old program (and variables) in the memory when it loads in the new one from tape.

Once a program has been loaded, the report **0 OK** will appear. The program can then be run or edited.

As mentioned in chapter 4, it is possible to buy pre-recorded programs (software) on tape. They must be specially written for the ZX Spectrum range (ie. the Spectrum, the Spectrum +, the Spectrum 128, the Spectrum +2 or the Spectrum +3). Different makes and models of computer have different ways of storing programs, so they cannot use each other's tapes.

If your tape has more than one program stored on the same side, then each program will have a name. You can choose the program you wish to load using the **LOAD** command - for instance, if the one you want is called **HELICOPTER** you could type

```
LOAD "helicopter"
```

The command **LOAD ""** means 'load the first program that the +3 comes across on tape'. This can be very useful if you cannot remember the name that you saved the program under! (Remember that this only works on tape - normally you cannot specify a blank filename.)

When there is no disk in drive A (or the disk contains no file called * or **DISK**), then the option **Loader** from the opening menu has the same action as **LOAD ""** from tape, and is much quicker to use - simply switch on (or reset) the +3 and press **ENTER**.

MERGE will operate in a similar way to that described for disk except, of course, that on tape you can use **MERGE ""** to mean 'merge the next file on tape'. Filenames in a **MERGE** command may conform to the less stringent limits for tape (ie. any combination of 10 characters including spaces).

If you have BASIC programs saved on tape (perhaps because you owned a previous Spectrum model), you will probably want to transfer them to disk to gain the advantage of faster loading. This should be relatively straightforward. Just use

```
LOAD "t:"
SAVE "a:"
```

then for each BASIC file on the tape, use

```
LOAD ""
```

which will load the next file from the tape into the +3's program memory. Once loaded, the file can be saved out to disk using

```
SAVE filename
```

Remember that files on disk must be given a filename which conforms to the limitations outlined at the beginning of this section.

If the BASIC programs have been saved with an automatic execution LINE, you will find that attempting to LOAD them will also run them. Obviously you don't want this so, for each program you wish to load, reset the computer, select +3 BASIC and type

```
MERGE ""
```

(rather than LOAD "")

If you have saved data (numeric or string) arrays, it should be an equally simple matter to LOAD them into memory from tape, then SAVE them to disk.

The only file types that may cause difficulty when you want to transfer them from tape to disk are CODE (and SCREEN$) files. To be able to transfer a file of this type you need to know at least two things about it:

1. The address it was saved from.
2. How many bytes it contains.

Tape catalog

This is where the final form of the CAT command comes in. If the file specification given is simply T: a special form of the CAT command comes into action. After you type

```
CAT "t:"
```

the +3 will wait for you to play a tape (the CAT "t:" operation can be abandoned by pressing **BREAK**). When the +3 finds a header on tape it will display the information (in the same form it was saved). This means that there will be a ten character filename in inverted commas. What follows the filename will depend upon the type of file - if it is a BASIC program the word (BASIC) will be displayed. If a LINE parameter was specified when the file was saved, this will also be shown. If the file holds data, then the word DATA followed by the array name will be displayed, and finally, if the file was saved using CODE (or SCREEN$ which is really just CODE 16384,6912) the word CODE will be printed followed by the start address and length that were specified when the file was saved.

Here is a sample display resulting from a `CAT "t:"` command, which may make this a little clearer

```
"simple      " (BASIC)
"execute     " LINE 10 (BASIC)
"numbers     " DATA f()
"words       " DATA c$()
"m/c         " CODE 30000,12345
"picture     " CODE 16384,6912
```

The last item was, in fact, saved using

```
SAVE "picture" SCREEN$
```

Just like the other forms of `CAT` its output can be directed to a printer using stream 3, ie

```
CAT #3,"t:"
```

(Streams are explained in part 22 of this chapter.) Note that the above `CAT #3,"t:"` command will not work unless a printer is connected to the **+3** and is on line. To abandon, press **BREAK**

From the above it can be seen that if you have loaded (using `MERGE ""`) a program containing an execution `LINE` parameter, the `CAT "t:"` display will identify that line number for you. You may then wish to save that program to disk using

```
SAVE filename LINE line number
```

so that the disk version of that program runs itself automatically.

It is the values for the `CODE` files that you will probably find most useful from the `CAT "t:"` display. Either note them down or print them out. Then rewind the tape so it is just before the header that has been read. Type

```
CLEAR start-1
```

where start is the value printed for the start address. Now type

```
LOAD "" CODE
```

When the file has loaded into memory and the `0 OK` report appears, the file can be saved to disk using

```
SAVE filename CODE start,length
```

This technique is only intended for transferring your own code files (where you may have forgotten what start and length values were used when you saved them). Note that using this method to copy commercial software may be a breach of copyright - check with the software author first.

There are several reasons why this simple scheme may *not work*:

1. The code, when loaded would overwrite some of the system variables (in the range 23296 (5B00h) to 23755 (5CC6h)). This upper address limit may vary - it is the value held in the system variable PROG (see part 25 of this chapter).

2. Attempting to load code that has no header (or that is protected in some other way) probably won't even produce any output from `CAT "t:"` and you certainly won't be able to use the BASIC `LOAD` command to load it.

3. If the code file is so long that it stretches right from the screen display area to the end of memory, then it will be possible to load it, but as soon as it has loaded, the machine will crash. This is because BASIC will have lost its stack.

Exercise

1. Practise the operations shown in this section until you are completely at fair with manipulating files to and from the disk, the RAMdisk and cassette unit (if connected).

Part 21
Printer operations

Subjects covered...

Parallel printers
Serial printers
LPRINT, LLIST
FORMAT
COPY

The +3 comes with an 8 bit Centronics parallel port and an RS232 serial port. Both are supported by built-in software enabling you to use virtually any printer. These features are usable only in +3 BASIC mode.

The printer must have either a Centronics compatible (parallel) or an RS232 (serial) interface, and if you want to reproduce pictures of the screen, then the printer must have an Epson compatible quadruple-density bit-image graphics mode (ESC L n m).

Make sure you have the correct lead to connect the printer to the +3. If in doubt, consult your Sinclair dealer.

For further information about which printer and connecting lead to purchase, together with details of the +3's **PRINTER** and **RS232** socket connections, see chapter [illegible] (Peripherals for your +3).

Parallel printers

When the +3 is first switched on it will assume that if a printer is present it will be connected to the (parallel) **PRINTER** socket. The hardware connection between computer and printer is relatively straightforward - though you must make sure that you don't connect the cable the wrong way up at the computer end (if the cable doesn't have a [illegible] key).

Once the connection has been made, the command

```
LPRINT "hello"
```

should produce some printed output. If not, check the connection and [illegible] sure that your printer is set to on line.

Once you have got your printer to print, you may [illegible] to the section ahead entitled General printing.

Serial printers

Unlike parallel printers, the connections between the *+3* and a serial (or RS232) printer will vary for different manufacturers' printers. Make sure that your dealer has provided a lead suitable for connecting your particular printer to the *+3*. A serial printer must be connected to the *+3*'s **RS232** socket and details of connections can be found in chapter 18 (Peripherals for your *+3*).

The *+3* always uses what is known as ***hardware flow control***, or ***hardware handshaking***. This means that it will not transmit characters until certain control signals from the printer have the right values. It is therefore very important that connections are made to the control lines of the *+3* as well as the transmit and receive data lines. If your printer does not support hardware handshaking then connect pins 4 and 5 of the *+3*'s **RS232** connector socket together. The drawback of not using hardware handshaking is that the odd character may be lost when transmitting a lot of data at high speed.

To get the *+3* and the printer communicating with each other they must both use the same ***baud rate***. The baud rate is the speed at which data is transferred between computer and printer. Although it is possible that your printer can be set to different baud rates it'll probably be easier to change the rate at the computer end. Somewhere in the printer's operating manual the baud rate will be specified - find this out and then set the *+3* to this rate using the command:

```
FORMAT LINE baud rate
```

For example:

```
FORMAT LINE 300
```

(You won't need to do this if your printer normally uses 9600 baud as the *+3* will assume this rate by default.)

As the *+3* usually expects to be operating with a parallel printer it will be necessary to use the command:

```
FORMAT LPRINT "R"
```

before the *+3* will successfully operate with a serial printer. (The R in the above command is short for RS232.)

The command to set the *+3* back to parallel (Centronics) mode is:

```
FORMAT LPRINT "C"
```

General printing

Once you have everything set up, you can use three BASIC commands to print things out. The first two **LPRINT** and **LLIST** are just like **PRINT** and **LIST** except that they use the printer instead of the TV screen. Note that the **Print** option from ***+3*** BASIC's edit menu has the same effect as **LLIST** but is included as an easier method of getting a listing.

Try this program for example:

```
10 LPRINT "This program..."'
20 LLIST 40
30 LPRINT '"...prints out the
   character set, ie..."'
40 FOR n=32 TO 255
50 LPRINT CHR$ n;
60 NEXT n
70 LPRINT
```

It's important to note that **LPRINT** and **LLIST** normally take care to screen out any embedded colour codes (and their parameters) before printing or listing anything. Embedded colour codes are a bit of a hangover from the old 48K Spectrum - when included in a string they set **INK PAPER** and so on. Printers on the [illegible] tend to use escape codes for completely different things like setting italics, underline, etc, so it would be quite dangerous to [illegible] colour codes to the printer and hope that nothing untoward happens. A side effect of this is that the ***+3*** will normally not be able to send ***escape control sequences*** to the printer. For example, suppose your printer expects an escape character (character 27) followed by "x";**CHR$(1)** to switch to its NLQ mode, you would normally use the command:

```
LPRINT CHR$ (27);"x"; CHR$ (1);"
This is in Near Letter Quality"
```

However in ***+3*** BASIC [illegible]

```
FORMAT LPRINT "U"
```

[illegible] ***+3*** [illegible] codes, but as [illegible] U [illegible] ***+3*** [illegible] words [illegible] ***tokens***. Likewise [illegible]

If you wish [illegible] ***+3*** [illegible]

```
FORMAT LPRINT "E"
```

[illegible] E [illegible] use **LLIST**. The ***+3*** [illegible] **FORMAT LPRINT "U"** [illegible] **FORMAT LPRINT "E"**

So to summarise

* If you want to send special characters (such as ESC) to your printer (in order to use different styles of printing), issue the command

```
FORMAT LPRINT "U"
```

before printing

* If you are writing or modifying a program, and want to get a listing on the printer, issue the command..

```
FORMAT LPRINT "E"
```

before listing the program

The third BASIC statement used with a printer - `COPY` prints out a copy of the TV screen. To demonstrate, go into the small screen (by selecting the `Screen` option from the edit menu) and type in the following command.

```
FOR n=1 TO 20: PRINT n,: NEXT n
```

The numbers 1 to 20 will be printed in the top part of the screen. Now type

```
COPY
```

The `COPY` command takes about 15-30 seconds to get started, so don't panic if nothing appears to happen immediately. After a while you'll see a copy of the screen reproduced on the printer. (If all you get from `COPY` is a lot of random characters on the printer, then it's likely that your printer isn't fully compatible.)

You can always stop printing at any time by pressing the **BREAK** key. Many printers have what's known as a ***buffer*** which stores text before printing. If your printer has a buffer, then pressing **BREAK** will not stop the printer immediately (although the ***+3*** will register the break at once).

Note that if the `COPY` command is stopped by pressing the **BREAK** key, the printer may be left in graphics mode (this will be indicated by subsequent `LPRINT` statements producing a mass of meaningless dots or printing each line of text partly over the previous line). In these circumstances switching the printer off then on [illegible] the easiest way to get things back to [illegible]

As well as the rather simple `COPY` command, which just produces a [illegible] the printer for each dot on the screen (whatever its colour [illegible] `COPY EXP`, which prints differing combinations [illegible] on the screen. To demonstrate, type in the [illegible]

```
10 FOR b=0 TO 1
20 BRIGHT b
30 FOR i=0 TO 6
40 FOR c=0 TO 31
50 PRINT INK i; i;
60 NEXT c
70 NEXT i
80 NEXT b
```

then switch to the bottom part of the screen (using the edit menus **Screen** option). Run the program (which displays two [illegible] lines of coloured numbers on the screen) then type in

```
COPY EXP
```

The printer output (or *dump*) from this command is slightly [illegible]ger than that from the standard **COPY** command (**EXP** is short for expanded). The command reproduces the coloured areas of the screen as different densities of black dots on the printer. (All 24 lines of the screen are reproduced.) Areas that have been printed with **BRIGHT 1** will appear lighter than areas printed normally (just as happens on the screen).

The drawback of the **COPY EXP** command is that it takes [illegible]nger time to print (about 10 minutes) but is ideally suited to dumping graphic pictures. The quicker **COPY** command, on the other hand, is a better bet if you wish to dump text only.

If the screen display to be dumped is predominantly black, then it will not only wear out your printer ribbon rather quickly, but also will probably also take [illegible] than [illegible]en that has large areas of white. To prevent this, the **COPY EXP** [illegible] be followed by the word **INVERSE** [illegible]

```
COPY EXP INVERSE
```

As the command suggests, the dump is printed in **INVERSE** (like a photographic negative) so that all the dark areas of the screen are printed-out light, and vice versa.

Note that **INVERSE** cannot be used after the simple **COPY** command — it only works with **COPY EXP**.

The dump produced by **COPY EXP** and **COPY EXP INVERSE** is designed to fit a sheet of A4 paper, however, some printers will not print within about an inch at either end of a sheet. If this problem occurs, then it is possible to reduce the size of the dump slightly by using the command

```
POKE 23419,8
```

This sets the number of 216ths inch used as a line feed at the end of each pass of the print head. It is set to 9 when the **+3** is first switched on. Once set [illegible] if the **NEW** command is used. By reducing this value, each pass of the print hea[illegible] overlay the previous pass. As a consequence, the quality of the dump reproduced will be degraded [illegible]ightly.

If you try to use any of the printer commands when there isn't a printer attached (or if the printer is off line) then the **+3** will stop dead while it patiently waits for the (non-existent) printer to respond. In such a case, pressing **BREAK** twice will bring the **+3** back to life.

Try this:

```
10 FOR n=31 TO 0 STEP -1
20 PRINT AT 31-n,n; CHR$ ( COD
   E "0"+n);
30 NEXT n
```

You will see a pattern of characters working down diagonally from the top right hand corner until it reaches the bottom of the screen, at which point the program asks if you want to scroll.

Now change `AT 31-n,n` in line 20 to `TAB n`. The program will have exactly the same effect as before.

Now change `PRINT` in line 20 to `LPRINT`. This time there will be no pause to scroll (this does not occur with the printer).

Now change `TAB n` back to `AT 31-n,n` still using `LPRINT`. This time you will get just a single line of symbols. The reason for the difference is that the output from `LPRINT` is not printed straight away, but is stored in the buffer until either one line's worth of printer output has accumulated, or something else flushes the buffer. Hence printing only takes place:

1. When the buffer is full.
2. After an `LPRINT` statement that does not end in a comma or semicolon.
3. When a comma, apostrophe or `TAB` item requires a new line.
4. At the end of a program, if there is anything left unprinted.
5. When you set the printer off line (this depends on your particular printer).

Number 3 above explains why our program with `TAB` works the way it does. As for `AT`, the line number is ignored, and the `LPRINT` position (like the `PRINT` position) is moved to the column number. An `AT` item can never cause a line to be sent to the printer.

Exercises

1. Make a printed graph of a sine wave by running the first (3 line) program in part 17 of this chapter, then using `COPY`.

2. Run the program at the beginning of part 16 of this chapter, and try both a `COPY EXP` and a `COPY EXP INVERSE`.

Part 22
Streams

Subjects covered...

Streams
Channels
FORMAT, OPEN, CLOSE

The **+3** can 'read' data from the keyboard by using **INPUT** and **INKEY$** and it can 'write' data onto the TV screen or a printer by using **PRINT** and **LPRINT**. However these commands are really a form of shorthand designed to protect the user from some of the computer's more complex features.

To the BASIC **PRINT** command, for example, the screen and the printer are no different. **PRINT "Rosanne"** really means 'take the characters which make up the word 'Rosanne' and send them somewhere else'. It's just convenient to use the screen most of the time. Likewise **LPRINT** usually sends data to the printer. In fact, what these commands really do is to send data to one of a number of ***channels***.

A channel is the way in which the computer communicates with its input and output devices. There are three channels normally available to BASIC. These are:

* The screen (called **S**)
* The keyboard (called **K**)
* The printer (called **P**)

Of these, the screen is an output-only device, the keyboard is both an input and output device, and the printer is either an output-only device (if it uses the parallel **PRINTER** socket) or an input and output device (if it uses the serial **RS232** socket). Outputting data to the keyboard might seem a funny idea but the computer uses the lower screen (like **INPUT** does) to display the characters.

To access a channel, it has to be ***open***. Opening a channel makes it ready to receive or produce data. A channel is opened by connecting it to a ***stream***. From BASIC you would use a command like:

```
OPEN #4,"p"
```

which means 'connect stream 4 to the printer channel'. Streams are convenient ways for the computer to switch between channels by referring to them as numbers. This idea makes it possible to write programs that can send information to any device without having to use different commands. (This is known as redirectable (or device-independent) I/O.)

This might seem over-complicated, and you may well wish to stick to the standard **PRINT** and **LPRINT** commands - that's why they're there, after all.

PRINT and LPRINT are really the same command. When BASIC is running, it has three streams normally open. Stream #1 is connected to the keyboard device (K), and is used by INPUT and INKEY$. Stream #2 is connected to the screen (S), and is used by PRINT and LIST. Stream #3 is connected to the printer (P), and is used by LPRINT and LLIST. All of these commands can be redirected to use another device by including a # followed by a current stream number, so

```
PRINT #1;"This is the lower screen"
```

will print the message on the lower screen. Similarly

```
PRINT #3;"Who needs LPRINT, Gladys?"
```

will use the printer. Conversely, LPRINT can behave like PRINT

```
LPRINT #2;"Confusing, or what?"
```

behaves just as if the LPRINT #2 were PRINT.

As they stand, these examples are fairly useless but serve to demonstrate a point. This sort of thing becomes useful if you want to write a program where the results might go either to the printer or the screen, like so

```
10 REM squares program for pri
   nter
20 INPUT "do you want to print
    the results?";a$
30 LET stream=2
40 IF a$="y" OR a$="Y" THEN LE
   T stream=3
50 FOR n=0 TO 10
60 PRINT #stream;n,n*n
70 NEXT n
```

The +3 can cope with 16 streams. As 3 are used by BASIC and 1 is used internally, this leaves you with 12. You can use these by

```
10 REM program to read data fr
   om RS232
20 FORMAT LINE 9600
30 FORMAT LPRINT "r"
40 OPEN #4,"p"
50 PRINT INKEY$ #4;
60 GO TO 50
```

which takes characters in from the RS232 interface and prints them onto the screen.

If you want to read in data from the RS232 into memory directly, you can replace line 50 with

```
POKE address, CODE (INKEY$ #4)
```

As we mentioned before, the screen and the parallel **PRINTER** socket can only be used by the ***+3*** for output. They cannot be used for input, and if you try `PRINT INKEY$ #2`, for example you'll receive an error report.

It is theoretically possible to redirect BASIC's normal output streams, so by using:

```
10 CLOSE #2
20 OPEN #2,"p"
```

all the `PRINT` output will go to the printer instead of the screen. (If you try to do this during editing the results will be unpredictable, so it's best left alone.)

On the standard ***+3*** system, streams and channels are of mostly academic interest. However certain peripherals and BASIC language extensions do use the stream system for more complex functions.

Part 23
IN and OUT

Subjects covered...

IN
OUT

The processor can read from (ROM and RAM) and write to (RAM) memory by using **PEEK** and **POKE**. The processor itself does not really care whether memory is ROM or RAM - it just thinks that there are 65536 memory addresses, and it can read a byte from each one (even if it's nonsense), and write a byte to each one (even if it gets lost). In a completely analogous way, there are 65536 of what are called ***I/O ports*** (standing for input/output ports). These are used by the processor for communicating with things like the keyboard or the printer, and also for controlling the extra memory and the sound chip. Some of them can be safely controlled from BASIC by using the **IN** function and the **OUT** command, but there are locations to which you must not write from BASIC as you will probably cause the system to crash, losing any program and data.

IN is a function like **PEEK**. Its form is

IN address

It has one argument - the port address, and its result is a byte read from that port.

OUT is a statement like **POKE**. Its form is

OUT address, value

, which writes the given value to the port with the given address. How the address is interpreted depends very much upon the rest of the computer. Quite often, many different addresses will mean the same. On the ***+3*** it is most sensible to imagine the address being written in binary, because the individual bits (each of which can have the value either 0 or 1) tend to work independently. There are 16 bits, which we shall refer to (using A for address) as:

A15, A14, A13, A12, A11, A10, A9, A8, A7, A6, A5, A4, A3, A2, A1, A0

Here, A0 is the 1s bit, A1 is the 2s bit, A2 is the 4s bit, and so on. Bits A0, A1, A2, A3 and A4 are the important ones. They are normally 1, but if any one of them is 0, then this tells the computer to do something specific. The computer cannot cope with more than one thing at a time, so no more than one of these five bits should be 0. Bits A6 and A7 are ignored, so if you are a wizard with electronics you can use them yourself. The best addresses to use are those that are 1 less than a multiple of 32, so that A0 to A4 are all 1. Bits A8, A9, and so on are sometimes used to give extra information, but mostly for the extra memory and sound.

The byte being written or read has 8 bits, and these are often referred to (using D for data) as :

D7, D6, D5, D4, D3, D2, D1, D0

Here follows a list of the port addresses used.

There is a set of input addresses that read the keyboard and the tape interface.

The keyboard is divided up into 8 half-rows of 5 keys each, viz

```
IN 65278   (FFFEh)   reads the half-row CAPS SHIFT to V
IN 65022   (FDFEh)   reads the half-row A to G
IN 64510   (FBFEh)   reads the half-row Q to T
IN 63486   (F7FEh)   reads the half-row 1 to 5 (and JOYSTICK 2)
IN 61438   (EFFEh)   reads the half-row 0 to 6 (and JOYSTICK 1)
IN 57342   (DFFEh)   reads the half-row P to Y
IN 49150   (BFFEh)   reads the half-row ENTER to H
IN 32766   (7FFEh)   reads the half-row (space) to B
```

(These addresses are 254+256x(255-2 ↑ n) as n goes from 0 to 7.)

Remember that digits followed by *h* signify hexadecimal numbers. If you don't understand these, refer to part 32 of this chapter.

In the byte read in, bits D0 to D4 stand for the five keys in the given half-row. D0 is for the outside key, and D4 is for the one nearest the middle. The bit is 0 if the key is pressed, 1 if it is not. D6 is set by the tape interface, and is effectively random if no tape data is present.

For JOYSTICK 1, bit 0 is fire, bit 1 is up, bit 2 is down, bit 3 is right and bit 4 is left. For JOYSTICK 2, bit 0 is left, bit 1 is right, bit 2 is down, bit 3 is up and bit 4 is fire. From BASIC these read as the number keys.

Port address 00FEh (254 decimal) in output drives the sound (D4) and the save signal to the tape interface (D3), and also sets the border colour (D2, D1 and D0).

Port addresses 00FEh (254), 00F7h (247) and 00EFh (239) are reserved.

Port address 7FFDh (32765) drives the extra memory. Executing an **OUT** to this port from BASIC will nearly always cause the computer to crash, losing any program and data. There is a fuller description of this port in part 24 of this chapter (under the heading 'Memory management'). This port is write only, ie. you cannot determine the current state of the paging by an **IN** instruction. This is why the BANKM system variable is always kept up to date with the last value output to this port.

Port address BFFDh (49149) drives the sound chip's data registers. Port address FFFDh (65533) in output writes a register address, and in input reads a register. Judicious use of these two registers can allow sounds to be generated whilst BASIC gets on with something else, but you should be aware that they also control **RS232/MIDI** and **AUX** interfaces.

Port address 0FFDh (4093) is used for the parallel (Centronics) interface (ie. **PRINTER**). When read using an **IN** instruction bit 0 shows the state of the BUSY signal produced by the printer. If the printer is off line or non-existent, then this bit will be 1. When this port is written to using **OUT**, it acts as the parallel port data register. In order to print a character it is necessary to wait until BUSY is 0, write the

character code to port 0FFDh (4093), and finally, take the STROBE bit in port 1FFDh (8189) low then back high again.

Port address 1FFDh (8189) controls several aspects of the ***+3*** Amongst other things, this port controls the ROM that is switched into the memory area from 0000h. 3FFFh (0 16383). As the port is write only, ***+3*** BASIC maintains a variable, BANK678, that holds the value last output to this port. It is therefore very unwise to **OUT** values directly to this port without first checking on the current state and setting/resetting only the bits you are interested in This is also the case for the port at 7FFDh (which holds its current state in BANKM) The bottom three bits (0..2) of this port (1FFDh) are used to switch RAM/ROM - further details can be found in part 24 of this chapter (under the heading 'Memory management') Bit 3 controls the disk motor (0 is off, 1 is on), though it should not be necessary to control the motor by writing to this port as there are +3DOS routines that will achieve the desired effect. Bit 4 is the parallel port STROBE which is active low - this means that to print the character that has been output to port 0FFDh (4093), the STROBE bit should be brought low and then returned to its normally high state.

Port address 2FFDh (12285) can be used to read the disk controller (µPD765A) chip's main status register This is unlikely to be very useful without an in-depth knowledge of how the chip operates.

Port address 3FFDh (16381) is the disk controller's data register This can be both read from and written to, but once again it is unlikely to be useful to the BASIC programmer Random **OUT** putting to this port will probably confuse the poor disk controller chip to such an extent that subsequent disk operations (like **LOAD** and **SAVE**) will be unreliable It is entirely possible that ill-informed experiments will corrupt your disks and lose your data - you have been warned!

Run this program to see how the keyboard works

```
10 FOR n=0 TO 7: REM half-row
   number
20 LET a=254+256*(255-2↑n)
30 PRINT AT 20,0; IN a: GO TO
   30
```

and play around by pressing keys (start with the half-row from **CAPS SHIFT** to **V**) When you finished with each half-row press **BREAK** and then type

```
NEXT n
```

The control, data and address busses are all exposed at the back of the ***+3*** on the **EXPANSION I/O** socket This means that you can do almost anything with a ***+3*** that you could with a raw Z80 chip (although sometimes, the computer's internal workings may get in the way)

See chapter 10 for a diagram and pin-out of the **EXPANSION I/O** socket

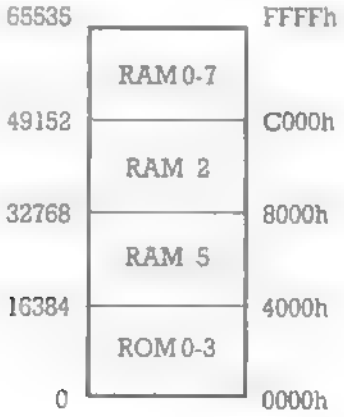
Part 24
The memory

Subjects covered...

PEEK
POKE
CLEAR
Memory management

Deep inside the *+3* everything is stored as ***bytes*** ie numbers between 0 and 255 (FFh). You may think you have stored away the price of Ruddles or the players names in the Arsenal football team, but in fact all the information has been converted into collections of bytes, and bytes are what the computer sees.

Each place where a byte can be stored has an address, which is a number between 0 (or 0000h) and 65535 (FFFFh). This means that an address can be stored as two bytes. You can think of the memory as a long row of numbered boxes, each of which can contain a byte. Not all the boxes are the same, however - the boxes from 4000h to FFFFh are ***RAM*** boxes, which means you can open the lid and alter the contents, but those from 0 to 3FFFh are ***ROM*** boxes, which have a glass lid that cannot be opened - you just have to read whatever was put into them when the computer was made. In the *+3* we have crammed in more than twice the amount of memory than can comfortably fit. While the processor can address 65536 bytes, there are in fact 131072 bytes of RAM and 65536 bytes of ROM making 196608 bytes (192K) in all! All this is hidden from the processor by the hardware using a process called paging. BASIC (and the processor) always sees the memory as 16K of ROM and 48K of RAM (or 64K of RAM with no ROM - though this latter combination is never used by BASIC).



The *+3* memory map

To inspect the contents of a box, we use the **PEEK** function. Its argument is the address of the box, and its result is the contents. For example, this program prints out the first 21 bytes in ROM (and their addresses):

```
10 PRINT "Address"; TAB 8; "By
   te"
20 FOR a=0 TO 20
30 PRINT a; TAB 8; PEEK a
40 NEXT a
```

All these bytes will probably be quite meaningless to you, but the processor chip understands them to be instructions telling it what to do.

To change the contents of a box (if it is RAM), we use the **POKE** command. Its form is:

POKE address,contents

where address and contents are numeric expressions. For example, if you type

```
POKE 31000,57
```

then the byte at address 31000 is given the new value 57. Now type

```
PRINT PEEK 31000
```

to prove this. (Try poking in other values, to show that there is no cheating.) The new value must be between −255 and +255; if it is negative, then 256 is added to it.

The ability to poke gives you immense power over the computer if you know how to wield it, and immense destructive possibilities if you don't. It is very easy (by poking the wrong value into the wrong address) to lose vast programs that took you hours to type in. Fortunately though, you won't do the computer any permanent damage.

We shall now take a more detailed look at how the RAM is used. Don't bother to read this unless you're really interested.

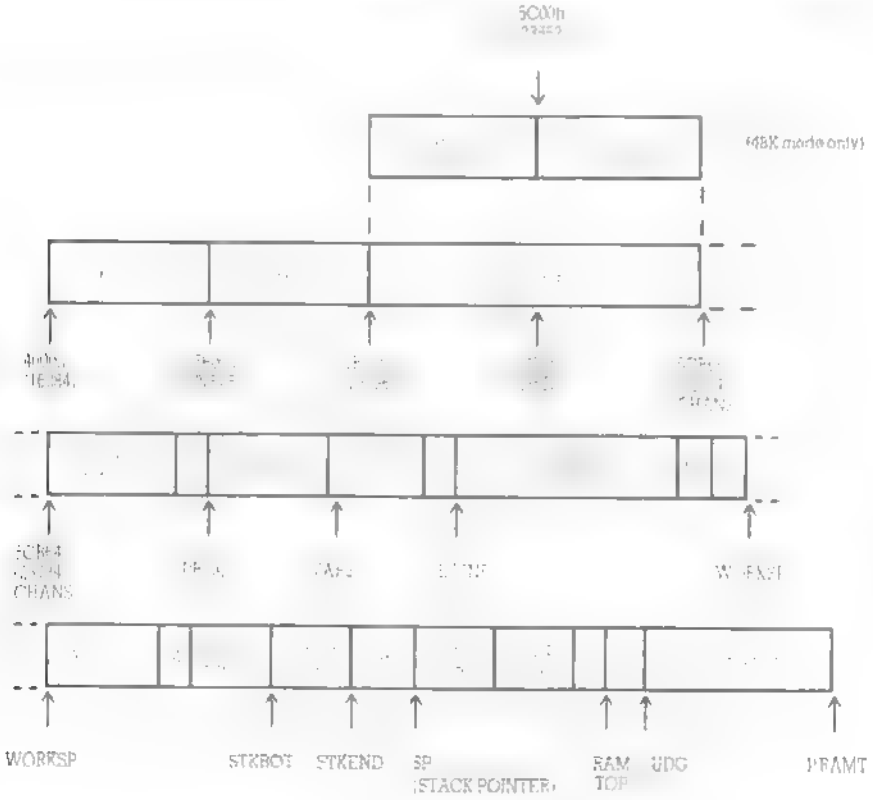
The memory is divided into different areas (shown in the diagram ahead) for storing different kinds of information. The areas are only large enough for the information that they actually contain, and if you insert some more at a given point (for instance by adding a program line or variable), then space is made by shifting up everything above that point. Conversely, if you delete information, then everything is shifted down.

The display file stores the contents of the TV screen. It is rather curiously laid out, so you probably won't want to **PEEK** or **POKE** in it. Each character position on the screen has an 8 x 8 grid of dots; each dot can be either 0 (paper) or 1 (ink), so by using binary notation we can store the pattern as 8 bytes—one for each row. However, these 8 bytes are not stored together. The corresponding columns in the 32 characters of a single line are stored together as a scan of 32 bytes, because this is what the electron beam in the TV needs as it scans from the left-hand side of the screen to the other. Since the

complete picture has 24 lines of 8 scans each, you might expect the total of 192 scans to be stored in order, one after the other - well, you'd be wrong! First come the top scans of lines 0 to 7, then the next scans of lines 0 to 7 and so on to the bottom scans of lines 0 to 7 then the same for lines 8 to 15, and again for lines 16 to 23. The upshot of all this is that if you're used to a computer that uses **PEEK** and **POKE** on the screen then you'll have to start using **SCREEN$** and **PRINT AT** instead (or **PLOT** and **POINT**).

The attributes are the colours and so on for each character position, using the format of **ATTR**. These are stored line by line in the order you'd expect.

The way that the computer organises its affairs changes slightly between 48 BASIC and ***+3*** BASIC mode. The area that was the printer buffer in 48 BASIC mode, is used for extra system variables in ***+3*** BASIC mode in much the same way as it was on the Spectrum ***+2***. The variables have changed, though.

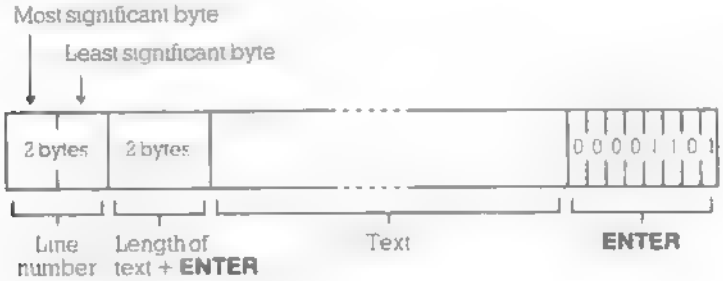


BASIC memory map

The system variables contain various pieces of information that tell the computer what sort of state it's in. They are listed fully in part 25 of this chapter, but for the moment, note that there are some (called CHANS, PROG, VARS, E LINE, and so on) that contain the addresses of the boundaries between the various areas in memory. These are not BASIC variables, and their names will not be recognised by the *+3*.

The channel information contains information about the input and output devices, namely the keyboard (together with the lower half of the screen), the upper half of the screen, and the printer.

Each line of BASIC program has the form

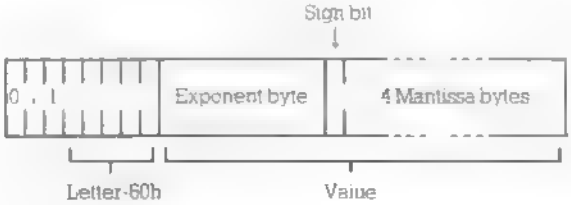


Note that, in contrast with all other cases of two-byte numbers in the Z80, the line number here is stored with its most significant byte first, ie in the order that you'd write them down.

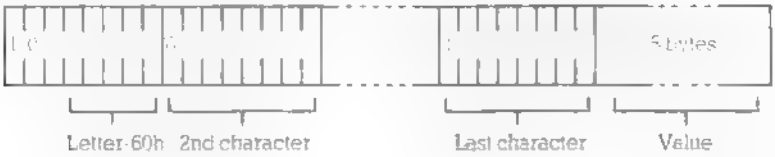
A numerical constant in the program is followed by its binary form, using the character CHR$ 14 followed by five bytes for the number itself.

The variables have different formats according to their different natures. The letters in the names should be imagined as starting off in lower case.

Number whose name is one letter only



Number whose name is longer than one letter



Array of numbers

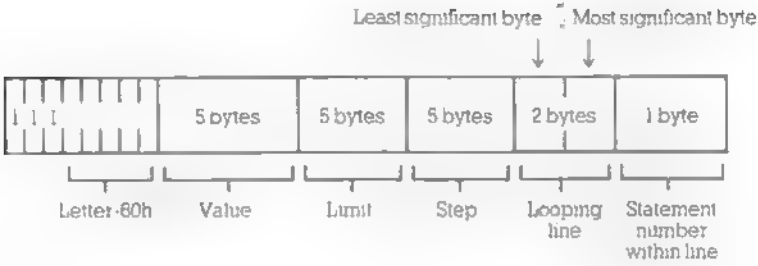
The order of the element is

First - the elements for which the first subscript is 1
Next - the elements for which the first subscript is 2
Next - the elements for which the first subscript is 3
and so on for all possible values of the first subscript

The elements with a given first subscript are ordered in the same way using the second subscript, and so on down to the last

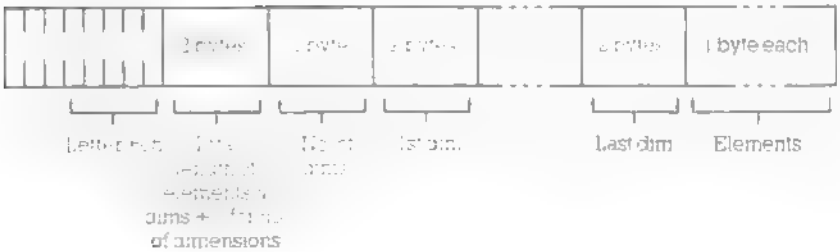
As an example the elements of the 3 x 6 array c in part 12 of this chapter are stored in the order
`c(1,1)` `c(1,2)` `c(1,3)` `c(1,4)` `c(1,5)` `c(1,6)` and
`c(2,1)` `c(2,2)` `c(2,6)` and `c(3,1)` `c(3,2)` `c(3,6)`

Control variable of a **FOR NEXT** loop



String

Array of characters



The calculator is the part of the BASIC system that deals with arithmetic and the numbers on which it is operating are held mostly in the calculator stack

The spare area contains the space so far unused.

The machine stack is the stack used by the Z80 processor to hold return addresses and so on.

The GO SUB stack was mentioned in part 5 of this chapter.

The byte 'pointed to' by RAMTOP has the highest address used by the BASIC system. Even NEW which clears the RAM, only does so as far as this - so it doesn't change the user-defined graphics. You can change the address RAMTOP by putting a number in a CLEAR statement, ie

CLEAR new RAMTOP

which does the following

1. Clears out all the variables
2. Clears the display file (like CLS does)
3. Resets the PLOT position to the bottom left hand corner
4. RESTORE s the DATA pointer
5. Clears the GO SUB stack and puts it at the new RAMTOP (assuming that this lies between the calculator and the physical end of RAM, otherwise it leaves RAMTOP where it was)

RUN also performs a CLEAR, although it never changes RAMTOP.

Using CLEAR in this way, you can either move RAMTOP up to make more room for the BASIC by overwriting the user-defined graphics, or you can move it down to make more RAM that is preserved from NEW. It can also be used to ensure that the machine stack is below BFEOh (49120) when intending to call +3DOS - this means that the stack will not have to be subsequently moved within your own machine code.

If you are in an experimental frame of mind you can also use CLEAR to explore the extra memory. CLEAR 49151 moves all of BASIC below the addresses that hold the switchable RAM paging. By using POKE 23388,16+n where n is a number between 0 and 7, you can make the computer switch in page n of the RAM. You will then be able to use PEEK and POKE in the normal way to examine and change the page. Beware - the extra pages are normally used by the system for disk and editor operations, so always reset the +3 after exploring in this way, before doing anything else.

Type NEW, select +3 BASIC then enter the command CLEAR 23825 to get some idea of what happens to the machine when it fills up.

If you then try to make the +3 compute (for example type in PRINT 1+1) you will see the report 4 Out of memory displayed. This means the computer has no more room for information. If you come up against this message while entering a large program, you will have to empty the memory slightly (delete a line or so) in order to control the computer.

Memory management

We mentioned earlier that there is rather more memory in the computer than the processor can deal with. While the processor can indeed address only 64K of memory at once, the extra memory can be slotted in and out of that 64K at will. Consider a TV set. Although it (and you) can only deal with one channel at a time, there are another three channels always there which can be selected with the right buttons. So, even though there's four times as much information as you can use at any one time, you can pick and choose which part is relevant.

It is much the same for the processor. By setting the right bits in an I/O port it can pick and choose which chunks of the 192K of memory it wants to use. For the majority of the time in BASIC it ignores most of the memory, but for games playing, having three times as much RAM is really rather useful. Look again at the *+3*'s memory map (shown at the beginning of this section). RAM pages 2 and 5 are always in the positions shown when BASIC is used, though there's no reason why they shouldn't be in the banked section (C000h to FFFFh) - however, it would be difficult to see any use for this.

The RAM banks are of two types: RAM pages 4 to 7 which are ***contended*** (meaning that they share time with the video circuitry) and RAM pages 0 to 3 which are ***uncontended*** (where the processor has exclusive use). Any machine code which has critical timing loops (such as music or communications programs) should keep all such routines in the uncontended banks. For example, executing NOPs in contended RAM will give an effective clock frequency of 2.66MHz as opposed to the normal 3.55MHz in uncontended RAM. This is a reduction in speed of about 25%.

The hardware switch normally used to select RAM is at I/O address 7FFDh (32765). The bit field for this address is as follows:

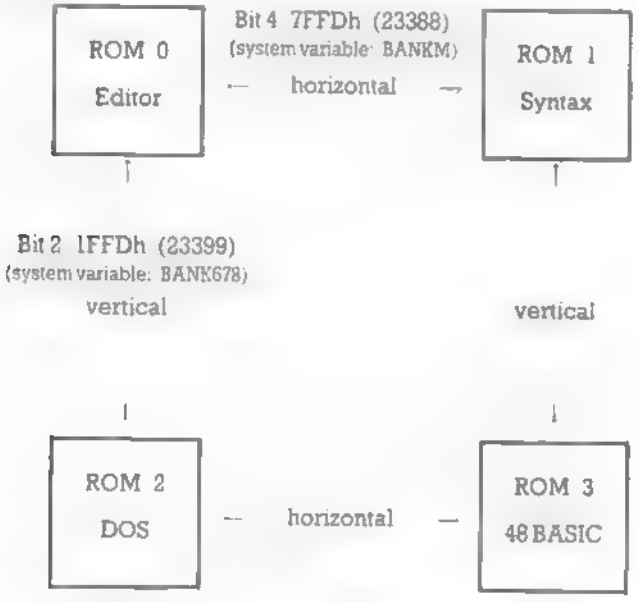
D0 - D2	- RAM select
D3	- Screen select
D4	- ROM select
D5	- Disable paging

D2 - D0 is a three bit number that selects which RAM page goes into the C000h to FFFFh slot. In BASIC, RAM page 0 is normally in situ. When editing or calling +3DOS routines, RAM page 7 is used for various buffers and scratchpads. D3 switches screens: screen 0 is held in RAM page 5 (normally beginning at 4000h) and is the one that BASIC uses. Screen 1 is held in RAM 7 (beginning at C000h) and can only be used by machine code programs. It is entirely feasible to set up a screen in RAM 7 and then page it out, this leaves the entire 48K free for data and program. Note that the *+3*'s `COPY` (file) command may well use buffers in the second screen area (corrupting a second screen which may be hidden there). D4 determines which ROM is paged into 0000h to 3FFFh (in combination with bit 2 of port 1FFDh - see below). D5 is a safety feature - once this bit has been set, no further paging operations will work. This is normally used when the the machine assumes a standard 48K Spectrum configuration and all the memory paging circuitry is locked out. It cannot be turned back into a 128K machine other than by switching off or pressing the **RESET** button, however, the sound chip can still be driven by `OUT`. If a 48K Spectrum game loaded from disk will not work, it is possible that by using the `SPECTRUM` command followed by `OUT 32765,48` (which locks bit 5 in this port) the game might then work.

The **+3** also uses I/O port 1FFDh for some ROM and RAM switching. The bit field for this address is as follows.

D0...D1	- ROM/RAM switching
D2	- Affects whether D0 ,D1 work on RAM/ROM
D3	- Disk motor
D4	- Parallel port strobe (active low)

When bit 0 is 0 bit 1 has no effect and bit 2 is a 'vertical' ROM switch (ie between ROM 0 and ROM 2 or between ROM 1 and ROM 3). Bit 4 in the port at 7FFDh is a 'horizontal' ROM switch (ie between ROM 0 and ROM 1, or between ROM 2 and ROM 3). The following diagram serves to show the various ROM switching possibilities.



Horizontal and vertical ROM switching

It is best to think of bit 4 in port 7FFDh and bit 2 in port 1FFDh combining to form a 2 bit number (0...3) which determines which ROM occupies the memory area 0000h - 3FFFh. Bit 4 of port 7FFDh is the least significant bit and bit 2 of 1FFDh is the most significant bit.

Bit 2 of 1FFDh (System variable: BANK678)	Bit 4 of 7FFDh (System variable: BANKM)	Switched ROM at 0000h...3FFFh
0	0	0
0	1	1
1	1	2
1	1	3

ROM switching (with Bit 0 of 1FFDh set to 0)

When bit 0 of port 1FFDh is set [illegible] bits 1 and 2 switch in various RAM combinations that occupy the full 64K address space. These are not used by +3 BASIC but are provided for authors of operating systems/games. When the +3 DOS BOOT routine is used, the bootstrap is loaded into the 4, 7, [illegible] 3 RAM page environment. The various +3 extra RAM page options are [illegible] follows

Bit 2 of 1FFDh	Bit 1 of 1FFDh	RAM pages used (0000h...3FFFh, 4000h...7FFFh, etc.)
0	0	0, 1, 2, 3
0	1	4, 5, 6, 7
1	0	4, 5, 6, 3
1	1	4, 7, 6, 3

Extended memory paging (with Bit 0 of 1FFDh set to 1)

Part 25
The system variables

Subjects covered...

POKE, PEEK

The bytes in memory from 5B00h (23296) to 5CB6h (23734) are set aside for specific uses by the system. There are a few routines (used to keep the paging in order) and some locations called *system variables*. You can peek these to find out various things about the system and some of them can be usefully poked. They are listed here with their uses.

There is quite a difference, as you might expect, between the system variables area in 48 BASIC mode and in *+3* BASIC mode. In 48 BASIC mode, all the variables and routines below 5C00h (23552) do not exist, instead there is a buffer between 5b00h (23296) and 5C00h (23552) which is used for controlling the printer. This was quite a popular location for small machine code routines on the old 48K Spectrum, and if any of these routines are tried in *+3* BASIC mode, the computer will invariably crash. Any old program that uses **PEEK**, **POKE** and **USR** is therefore a safer bet if it is run in 48 BASIC mode (although it can be entered in *+3* BASIC mode and transferred using the **SPECTRUM** command). If there is a chance that a program might inadvertently address the added I/O ports of the *+3* then **OUT 32765,48** will set bit 5 in port 7FFDh to disable further use of the added ROM/RAM switching.

Although system variables have names, you should not confuse them with the words and names used in BASIC. The computer will not recognise the names as referring to system variables, they are given solely as mnemonics for we humans.

The abbreviations in column 1 of the table ahead have the following meanings:

X - The variables should not be poked because the system might crash.

N - Poking the variables will have no lasting effect.

R - Routine entry point. Not a variable.

The number in column 1 is the number of bytes in the variable or routine. For a two-byte word, the first byte is the least significant - the reverse of what you might expect. So to poke a value v into a two-byte variable at address n, use:

```
POKE n,v-256* INT (v/256)
POKE n+1, INT (v/256)
```

.and to peek its value, use the expression:

```
PRINT PEEK n+256* PEEK (n+1)
```

NOTES	ADDRESS HEX (DECIMAL)	NAME	CONTENTS
R16	5B00h (23296)	SWAP	Paging subroutine
R17	5B10h (23312)	STOO	Paging subroutine. Entered with interrupts already disabled and AF, BC on the stack
R9	5B21h (23329)	YOUNGER	Paging subroutine
R16	5B2Ah (23338)	REGNUOY	Paging subroutine.
R24	5B3Ah (23354)	ONERR	Paging subroutine.
X2	5B52h (23378)	OLDHL	Temporary register store while switching ROMs
X2	5B54h (23380)	OLDBC	Temporary register store while switching ROMs
X2	5B56h (23382)	OLDAF	Temporary register store while switching ROMs
N2	5B58h (23384)	TARGET	Subroutine address in ROM 3
X2	5B5Ah (23386)	RETADDR	Return address in ROM 1.
X1	5B5Ch (23388)	BANKM	Copy of last byte output to I/O port 7FFDh (32765). This port is used to control the RAM paging (bits 0..2), the 'horizontal' ROM switch (0⟷1 and 2⟷3 - bit 4), screen selection (bit 3) and added I/O disabling (bit 5). This byte must be kept up to date with the last value output to the port if interrupts are enabled
X1	5B5Dh (23389)	RAMRST	RST 8 instruction. Used by ROM 1 to report old errors to ROM 3
N1	5B5Eh (23390)	RAMERR	Error number passed from ROM 1 to ROM 3. Also used by **SAVE/LOAD** as temporary drive store
2	5B5Fh (23391)	BAUD	RS232 bit period in T states/26. Set by **FORMAT LINE**
N2	5B61h (23393)	SERFL	Second-character-received-flag, and data
N1	5B63h (23395)	COL	Current column from 1 to width
1	5B64h (23396)	WIDTH	Paper column width. Defaults to 80
1	5B65h (23397)	TVPARS	Number of inline parameters expected by RS232
1	5B66h (23398)	FLAGS3	Various flags. Bits 0, 1, 6 and 7 unlikely to be useful. Bit 2 is set when tokens are to be expanded on printing. Bit 3 is set if print output is RS232. The default (at reset) is Centronics. Bit 4 is set if a disk interface is present. Bit 5 is set if drive B is present.

NOTES	ADDRESS HEX (DECIMAL)	NAME	CONTENTS
X1	5B67h (23399)	BANK678	Copy of last byte output to I/O port 1FFDh (8189). This port is used to control the +3 extended RAM and ROM switching (bits 0..2), if bit 0 is 0 then bit 2 controls the vertical ROM switch 0⟷2 and 1⟷3), the disk motor (bit 3) and Centronics strobe (bit 4). This byte must be kept up to date with the last value output to the port if interrupts are enabled.
N1	5B68h (23400)	XLOC	Holds X location when using the unexpanded COPY command.
N1	5B69h (23401)	YLOC	Holds Y location when using the unexpanded COPY command.
X2	5B6Ah (23402)	OLDSP	Old SP (stack pointer) when TSTACK in use.
X2	5B6Ch (23404)	SYNRET	Return address for ONERR.
5	5B6Eh (23406)	LASTV	Last value printed by calculator.
2	5B73h (23411)	RC LINE	Current line being renumbered.
2	5B75h (23413)	RC START	Starting line number for renumbering. The default value is 10.
2	5B77h (23415)	RC STEP	Incremental value for renumbering. The default is 10.
1	5B79h (23417)	LODDRV	Holds T if LOAD VERIFY MERGE are from tape, otherwise holds A B or M.
1	5B7Ah (23418)	SAVDRV	Holds T if SAVE is to tape, otherwise holds A B or M.
1	5B7Bh (23419)	DUMPLI	Holds the number of 1/216ths used for line [illegible] COPY EXP. This is normally set [illegible] problems are experienced fitting [illegible] onto a sheet of A4 paper. POKE this [illegible] with 6. This will reduce the size of [illegible] and improve the aspect ratio slightly. (The quality of the dump will be marginally degraded, however.)
N8	5B7Ch (23420)	STRIP1	Strip one bitmap.
N8	5B84h (23428)	STRIP2	Strip two bitmap. This extends to 5B8Eh (23436).
X115	5BFFh (23551)	TSTACK	Temporary stack grows down from here. [illegible] is switched in at [illegible] ing the editor [illegible] go down to 5B8Ch (and across STRIP1 and STRIP2 if necessary). This guarantees at least 115 bytes of stack when BASIC calls +3DOS.

NOTES	ADDRESS HEX (DECIMAL)	NAME	CONTENTS
N8	5C00h (23552)	KSTATE	Used in reading the keyboard
N1	5C08h (23560)	LAST K	Stores newly pressed key
1	5C09h (23561)	REPDEL	Time (in 50ths of a second) that a key must be held down before it repeats. This starts off at 35, but you can POKE in other values
1	5C0Ah (23562)	REPPER	Delay (in 50ths of a second) between successive repeats of a key held down -initially 5
N2	5C0Bh (23563)	DEFADD	Address of arguments of user defined function (if one is being evaluated) otherwise 0.
N1	5C0Dh (23565)	K DATA	Stores 2nd byte of colour controls entered from keyboard
N2	5C0Eh (23566)	TVDATA	Stores bytes of colour AT and TAB controls going to TV
X38	5C10h (23568)	STRMS	Addresses of channels attached to streams
2	5C36h (23606)	CHARS	256 less than address of character set (which starts with space and carries on to ©). Normally in ROM, but you can set up your own in RAM and make CHARS point to it
1	5C38h (23608)	RASP	Length of warning buzz
1	5C39h (23609)	PIP	Length of keyboard click
1	5C3Ah (23610)	ERR NR	1 less than the report code. Starts off at 255 (for 1 - so PEEK 23610 gives 255
X1	5C3Bh (23611)	FLAGS	Various flags to control the BASIC system
X1	5C3Ch (23612)	TV FLAG	Flags associated with the TV
X2	5C3Dh (23613)	ERR SP	Address of item on machine stack to be used as error return
N2	5C3Fh (23615)	LIST SP	Address of return address from automatic listing
N1	5C41h (23617)	MODE	Specifies K L C E or G cursor
2	5C42h (23618)	NEWPPC	Line to be jumped to
1	5C44h (23620)	NSPPC	Statement number in line to be jumped to. Poking first NEWPPC and then NSPPC forces a jump to a specified statement in a line
2	5C45h (23621)	PPC	Line number of statement currently being executed
1	5C47h (23623)	SUBPPC	Number within line of statement being executed
1	5C48h (23624)	BORDCR	Border colour multiplied by 8; also contains the attributes normally used for the lower half of the screen.
2	5C49h (23625)	E PPC	Number of current line (with program cursor)

NOTES	ADDRESS HEX (DECIMAL)	NAME	CONTENTS
X2	5C4Bh (23627)	VARS	Address of variables
N2	5C4Dh (23629)	DEST	Address of variable in assignment
X2	5C4Fh (23631)	CHANS	Address of channel data
X2	5C51h (23633)	CURCHL	Address of information currently being used for input and output
X2	5C53h (23635)	PROG	Address of BASIC program.
X2	5C55h (23637)	NXTLIN	Address of next line in program
X2	5C57h (23639)	DATADD	Address of terminator of last **DATA** item
X2	5C59h (23641)	E LINE	Address of command being typed in.
2	5C5Bh (23643)	K CUR	Address of cursor
X2	5C5Dh (23645)	CH ADD	Address of the next character to be interpreted the character after the argument of **PEEK**, or the NEWLINE at the end of a **POKE** statement
2	5C5Fh (23647)	X PTR	Address of the character after the ▮ marker
X2	5C61h (23649)	WORKSP	Address of temporary work space
X2	5C63h (23651)	STKBOT	Address of bottom of calculator stack
X2	5C65h (23653)	STKEND	Address of start of spare space
N1	5C67h (23655)	BREG	Calculator s B register
N2	5C68h (23656)	MEM	Address of area used for calculator's memory (usually MEMBOT, but not always)
1	5C6Ah (23658)	FLAGS2	More flags (Bit 3 set when **CAPS SHIFT** or **CAPS LOCK** is on.)
X1	5C6Bh (23659)	DF SZ	The number of lines (including one blank line) in the lower part of the screen
2	5C6Ch (23660)	S TOP	The number of the top program line in automatic listings
2	5C8Eh (23662)	OLDPPC	Line number to which **CONTINUE** jumps
1	5C70h (23664)	OSPCC	Number within line of statement to which **CONTINUE** jumps
N1	5C71h (23665)	FLAGX	Various flags
N2	5C72h (23666)	STRLEN	Length of string type destination in assignment
N2	5C74h (23668)	T ADDR	Address of next item in syntax table (very unlikely to be useful)
2	5C76h (23670)	SEED	The seed for **RND**. This is the variable that is set by **RANDOMIZE**
3	5C78h (23672)	FRAMES	3 byte (least significant byte first), frame counter incremented every 20mS.
2	5C7Bh (23675)	UDG	Address of first user-defined graphic. You can change this, for instance, to save space by having fewer user-defined graphics.

NOTES	ADDRESS HEX (DECIMAL)	NAME	CONTENTS
1	5C7Dh (23677)	COORDS	X-coordinate of last point plotted.
1	5C7Eh (23678)		Y-coordinate of last point plotted.
1	5C7Fh (23679)	P POSN	33-column number of printer position.
1	5C80h (23680)	PR CC	Least significant byte of address of next position for **LPRINT** to print at (in printer buffer)
1	5C81h (23681)		Not used
2	5C82h (23682)	ECHO E	33-column number and 24-line number (in lower half) of end of input buffer
2	5C84h (23684)	DF CC	Address in display file of **PRINT** position.
2	5C86h (23686)	DF CCL	Like DF CC for lower part of screen.
X1	5C88h (23688)	S POSN	33-column number for **PRINT** position.
X1	5C89h (23689)		24-line number for **PRINT** position.
X2	5C8Ah (23690)	SPOSNL	Like S POSN for lower part
1	5C8Ch (23692)	SCR CT	Counts scrolls - it is always 1 more than the number of scrolls that will be done before stopping with **scroll?** If you keep poking this with a number bigger than 1 (say 255), the screen will scroll on and on without asking you
1	5C8Dh (23693)	ATTR P	Permanent current colours, etc., (as set up by colour statements)
1	5C8Eh (23694)	MASK P	Used for transparent colours etc. Any bit that is 1 shows that the corresponding attribute bit is taken not from ATTR P, but from what is already on the screen.
N1	5C8Fh (23695)	ATTR T	Temporary current colours, etc., (as set up by colour items)
N1	5C90h (23696)	MASK T	Like MASK P, but temporary
1	5C91h (23697)	P FLAG	More flags.
N30	5C92h (23698)	MEMBOT	Calculator's memory area - used to store numbers that cannot conveniently be put on the calculator stack.
2	5CB0h (23728)	NMIADD	Holds the address of the users NMI service routine NOTE - On previous machines, this did not work correctly and these two bytes were documented as 'Not used.' Programs that used these two bytes for passing values may need to be modified
2	5CB2h (23730)	RAMTOP	Address of last byte of BASIC system area.
2	5CB4h (23732)	P RAMT	Address of last byte of physical RAM

Part 26
Using machine code

Subjects covered...

USR with numeric argument

This section is written for those who understand Z80 machine code, ie the set of instructions that the Z80 processor chip uses. If you do not, but would like to, there are plenty of books about it. You should get one called something along the lines of Z80 machine code (or assembly language) for the absolute beginner, and if it mentions the *+3* or other computers in the ZX Spectrum range, so much the better.

Machine code programs are normally written in ***assembly language***, which, although cryptic, is not too difficult to understand with practice. You can see the assembly language instructions in part 28 of this chapter. However, to run them on the *+3*, you need to code the program into a sequence of bytes - then called machine code. This translation is usually done by the computer itself using a program called an ***assembler***. There is no assembler built in to the *+3*, but you will be able to buy one on disk or tape. Failing that, you will have to do the translation yourself, provided that the program is not too long.

Let's take as an example the program

```
ld bc, 99
ret
```

, which loads the BC register pair with 99. This translates into the four machine code bytes 1, 99, 0 (for `ld bc, 99`) and 201 (for `ret`). (If you look up codes 1 and 201 in part 28 of this chapter, you will find that 1 corresponds to *ld bc, NN* - where *NN* stands for any two-byte number, and 201 corresponds to *ret*.)

When you have got your machine code program, the next step is to get it into the computer - (an assembler would probably do this automatically). You need to decide whereabouts in memory to locate it - the best thing is to make extra space for it between the BASIC area and the user-defined graphics.

If you type

```
CLEAR 65267
```

this will give you a space of 100 (for good measure) bytes starting at address 65268.

To put in the machine code program, you would run a BASIC program something like

```
10 LET a=65268
20 READ n: POKE a,n
30 LET a=a+1: GO TO 20
40 DATA 1,99,0,201
```

(This will stop with the report `E Out of DATA` when it has filled in the four bytes you specified.)

To run the machine code, you use the function USR - but this time with a numeric argument, ie the starting address. Its result is the value of the BC register on return from the machine code program, so if you type

```
PRINT USR 65268
```

you will get the answer 99

The return address to BASIC is stacked in the usual way, so return is by a Z80 *ret* instruction. You should not use the IY and I registers in a machine code routine that expects to use the BASIC interrupt mechanism. If you are writing a program that might eventually run on an older Spectrum (up to and including the *+2*) you should not load I with values between 40h and 7Fh (even if you never use IM 2). Values between C0h and FFh for I should also be avoided if contended memory (ie RAM 4 to 7) is to be paged in between C000h and FFFFh. This is due to an interaction between the video controller and the Z80 refresh mechanism, and can cause otherwise inexplicable crashes, screen corruption or other undesirable effects. Thus, you should only vector IM 2 interrupts to between 8000h and BFFFh unless you are very confident of your memory mapping (or you are only going to run your program on the *+3* where this problem does not exist).

The system variable at 5CB0h (23728) was documented on previous models of the Spectrum as 'Not used'. It is now used on the *+3* as an NMI jump vector. If an NMI occurs, this address is checked. If it contains 0, then no action is taken. However, for any other (non-zero) value, a jump will be made to the address given by this variable. NMIs must not occur while the disk system is active.

There are a number of standard pitfalls when programming a banked system such as the *+3* from machine code. If you are experiencing problems, check that your stack is not being paged out during interrupts, and that your interrupt routine is always where you expect it to be (it is advisable to disable interrupts during paging operations). It is also recommended that you keep a copy of the current bank register setting in unpaged RAM somewhere as the ports are write-only. BASIC and the editor use the system variables BANKM and BANK678 for 7FFDh and 1FFDh respectively.

If you call +3DOS routines, remember that interrupts should be enabled upon entry to the routines. Remember also that the stack must be below BFE0h (49120) and above 4000h (16384), and that there must be at least 50 words of stack space available.

You can save your machine code program easily enough with (for example)

```
SAVE "name" CODE 65268,4
```

On the face of it, there is no way of saving the program so that when loaded it automatically runs itself. However, you can get round this by using the short BASIC program

```
10 LOAD "name" CODE 65268,4
20 PRINT USR 65268
```

which should also be saved (as a separate program) using the command (for example)

```
SAVE "loader" LINE 10
```

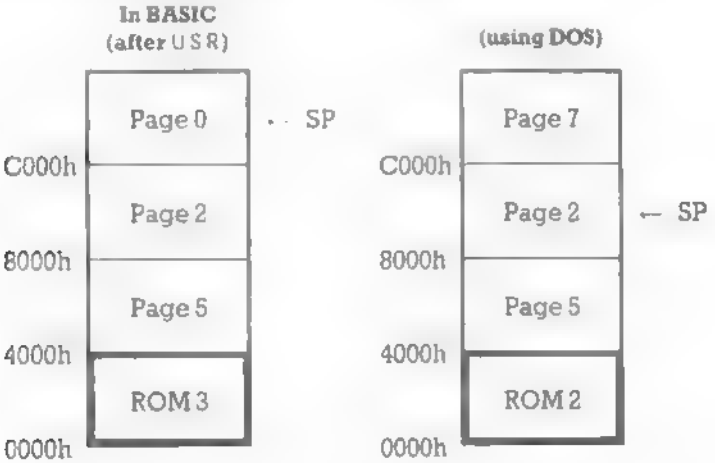
Then you may run the machine code from BASIC using the single command

```
LOAD "loader"
```

which loads and automatically runs the BASIC program which in turn loads and runs the machine code

Calling +3DOS from BASIC

When BASIC's **USR** function is used, the code it references is entered with the memory configured as illustrated below (left), ■ the ROM switched in at the bottom of memory ■ the address range (0000h 3FFFh) is ROM 3 (the 48 BASIC ROM) The RAM page at the top of memory is page 0 and the machine stack resides in this area (unless the **CLEAR** command has been used to reduce it to somewhere below C000h) As explained in part 27 of this chapter (which describes the +3DOS routines), DOS can only be called with RAM page 7 switched in at the top of memory the stack held somewhere in the range 4000h BFE0h and ROM 2 (the DOS ROM) switched in at the bottom of memory (0000h 3FFFh) This configuration is illustrated below (right)



Consequently it will be necessary to switch both ROM and RAM and move the stack before and after calling one of the entries in the DOS jump table The following very simple example shows one way of achieving the desired set up in order to call DOS CATALOG

If BASIC's **CLEAR** command has been used so that the BASIC stack is below BFE0h (49120), then it is not necessary to move the stack However we have done so in the following example to demonstrate the technique when this is not the case

A simple example to call DOS CATALOG

```
        org     7300h

mystack         equ     9FFFh       ;arbitrary value picked to be betw. BFEDh and above 4000h
stakstb         equ     9200h       ;somewhere to put BASIC's stack pointer
bankm           equ     5B5Ch       ;system variable that holds the last value output to 7FFDh
portf           equ     7FFDh       ;address of ROM/RAM switching port in I/O map
catbuff         equ     8200h       ;somewhere for DOS to put its catalog
dos catalog     equ     B1ECh       ;the DOS routine to call

demo:

        di                          ;advise to switch RAM/ROM without disabling interrupts
        ld      (stakstb),sp        ;save BASIC's stack pointer
        ld      bc,portf            ;the horizontal ROM switch/RAM switch I/O address
        ld      a,(bankm)           ;system variable that holds current switch state
        res     4,a                 ;move right to left in horizontal ROM switch (3 to 2)
        or      7                   ;switch in RAM page 7
        ld      (bankm),a           ;must keep system variable up to date (very important)
        out     (c),a               ;make the switch
        ld      sp,mystack          ;make sure stack is above 4000h and below BFEDh
        ei                          ;interrupts can now be enabled


;
; The above will have switched in the DOS ROM and RAM page 7. The stack has also been
; located in a "safe" position for calling DOS
;
; The following is the code to set up and call DOS CATALOG. This is where your own
; code would be placed.
;

        ld      ix,catbuff          ;somewhere for DOS to put the catalog
        ld      de,catbuff+1
        ld      bc,1024             ;our buf is actually 64x13+13 = 845
        ld      (hl),0
        ldir                        ;make sure at least first entry is zeroised
        ld      b,64                ;the number of entries in the buffer
        ld      c,1                 ;include system files in the catalog
        ld      de,catbuff          ;the location to be filled with the disk catalog
        ld      hl,"*.*"            ;the file name "*.*"
        call    dos catalog         ;call the DOS entry
        push    af                  ;save flags and possible error number returned by DOS
        pop     hl
        ld      (dosret),hl         ;put it where it can be seen from BASIC
        ld      c,b                 ;move number of files in catalog to low byte of BC
        ld      b,0                 ;this will be returned in BASIC by the USR function


;
; If the above worked, then BC holds number of files in catalog, the "catbuff" will
; be filled with the alphanumerically sorted catalog and the carry flag bit in "dosret"
; will be set. This will be peeked from BASIC to check if all went well.
;
; Having made the call to DOS, it is now necessary to undo the ROM and RAM switch and
; put BASIC's stack back to where it was on entry. The following will achieve this.
;
```

```
        di                          ;about to ROM/RAM switch so be careful
        push    ix                  ;save number of files
        ld      bc,port1            ;i.e address of horizontal ROM/RAM switch
        ld      a,(bankm)           ;get current switch state
        set     4,a                 ;move left to right (ROM 2 to ROM 3)
        and     FBh                 ;also want RAM page 0
        ld      (bankm),a           ;update the system variable (very important)
        out     (c),a               ;make the switch
        pop     bc                  ;get back the saved number of files in catalog
        ld      sp,(stakstp)        ;get BASIC's stack back
        ret                         ;return to BASIC, value in BC is returned to USR

stargstart:

        defb    "*.*",FFh           ;the file name, must be terminated with FF

dosret:

        defw    0                   ;a variable to be peeked from BASIC to see if it worked
```

As some of you may not have an assembler available the following is a BASIC program that pokes the above code into memory calls it and then uses the value returned by the **USR** function and the contents of dosret to print a very simple catalog of the disk

```
 10 LET sum=0
 20 FOR i=28672 TO 28758
 30 READ n
 40 POKE i,n : LET sum=sum+n
 50 NEXT i
 60 IF sum <> 9387 THEN PRINT "
    Error in DATA" : STOP
 70 LET x=USR 28672
 80 IF INT ( PEEK (28757)/2)= P
    EEK (28757)/2 THEN PRINT "D
    isk error "; PEEK (28758):
    STOP
 90 IF x=1 THEN PRINT "No files
     found": STOP
100 FOR i=0 TO x-2
110 FOR j=0 TO 10
120 PRINT CHR$ ( PEEK (32781+i*
    13+j));
130 NEXT j
140 PRINT
150 NEXT i
160 DATA 243,237,115,0,144,1,25
    3,127,58,92,91,203,167,246,
    7,50,92,91,237,121,49,255,1
    59,251
```

```
170 DATA 33,0,128,17,1,128,1,0,
    4,54,0,237,176,6,64,14,1,17
    ,0,128,33,81,112,205,30,1,2
    45,225,34,85,112,72,6,0
180 DATA 243,197,1,253,127,58,9
    2,91,203,231,230,248,50,92,
    91,237,121,193,237,123,0,14
    4,201
190 DATA 42,46,42,255,0,0
```

The addresses picked for the above code and its data areas are completely arbitrary. However, it is a good idea to keep things in the central 32K wherever possible so as not to run into the pitfall of accidentally switching out a vital variable or piece of code.

If interrupts are to be enabled (as is the case in the above example) it is imperative that the system is kept up to date about the latest ROM switch. This means that the user must make the BANK678 system variable reflect the last value output to the port at 1FFDh. As shown by the above example, the general technique is to take a copy of the variable in A, set/reset the relevant bits, update the system variable, then make the switch with an `OUT` instruction. Interrupts must be disabled while the system variable does not reflect the current state of the port. The port at 1FFDh doesn't just control the ROM switch, so setting the variable to absolute values would be very unwise. Using AND/OR with a bit mask or SET/RES instructions is the preferred method of updating the variable.

Just as BANK678 reflects the last value output to 1FFDh, BANKM should also be kept up to date with the last value output to 7FFDh. Again, it is unwise to use absolute values, as the port is used for other purposes. For example, the bottom 3 bits of the port are used to select the RAM page that is switched into the memory area C000h - FFFFh (this is also shown in the above example). Naturally, when more than one bit is to be set/reset, a bit mask used with OR/AND is the more efficient method. Note that RAM paging was described in the section entitled 'Memory management' in part 24 of this chapter.

The above was a very simple example of calling DOS routines. The following shows one or two extra techniques that you may find useful. However, if you are not already familiar with assembler programming, it might be better to skip this example.

Although part 20 of this chapter suggested that the opening menu's `Loader` option first looks for a file called `*` and then one called `DISK` before trying to load the first file from tape - this isn't exactly the whole story. The first operation actually tries to load a ***bootstrap*** sector from the disk in drive A. The sector on side 0, track 0, sector 1 will be used as a loader (bootstrap) if the system finds that the 8 bit checksum of the sector is 3. The following program ensures that the checksum of 512 bytes conforms to this requirement, then writes the information to the disk in the correct position. Once a disk has been modified in this way, the `Loader` option can be used to automatically load and run the disk. Alternatively, the BASIC command `LOAD "*"` can be used.

This example was developed using M80 on a CP/M based machine - so the method used to ensure that the code is assembled relative to the correct address might be different from that used by your own assembler

```
;
; Simple example program to write a boot sector to the disk in drive A:.
;
; by Cliff Lawson
; copyright (c) AMSTRAD Plc. 1987
;


        .z80                            ;ignore this if not using M80


bank1           equ     2'7f0h          ;"horizontal" and RAM switch port
bank2           equ     0505ch          ;associated system variable
bank3           equ     0187f0h         ;"vertical" switch port
bank6/8         equ     0506fh          ;associated system variable


select          equ     0'001h          ;BASIC routine to open stream
dos rel supp    equ     0151h           ;
dd write sector equ     0166h           ;see part 27 of this chapter
dd login        equ     0175h           ;

        org     0

        .phase  0f000h


;
; (This allows M80 to generate a .COM file that has addresses relative to 7000h
; Assemble with "M80 =prog" and [illegible] 32 [illegible] "[illegible]:prog/m:0/z/e"
; This can be needed with COPY[illegible] SPECTRUM FORMAT and loaded with
; LOAD...CODE 28672.
;
; A different technique will probably be required for other assemblers.
;
start:

        ld      (oldstack),sp           ;save BASIC's stack pointer
        ld      sp,mystack              ;put stack below switched RAM pages
        push    af                      ;save [illegible] stack for the moment

        ld      a,"A"                   ;[illegible]
        ld      iy,dos rel addr         ;make IY point to XDPB A: (necessary for calling DD
        call    nodos                   ;routines)

        ld      c,0                     ;log in disk in unit 0 so that writing sectors
        push    ix                      ;won't say "disk has been changed"
        ld      ix,dd login
        call    nodos
        pop     ix

        ld      hl,bootsector
        ld      bc,512                  ;[illegible]
        xor     a
        ld      (bootsector+15),a       ;reset checksum for starters
        exp     e,a                     ;E = 11 mode 5 drt sys


        ld      a,w
        add     a,(hl)                  ;[illegible] .COD takes 8 bit checksum [illegible] 512 bytes
        ld      e,a                     ;[illegible] = E
```

```
        dec     bc
        ld      a,b
        or      c
        jr      nz,ckloop

        ld      a,e                     ;A now has 8 bit checksum of the sector
        cpl                             ;ones comp (xor ff will give negative value)
        add     a,4                     ;add 3 to make sum = 3 + 1 to make twos complement
        ld      (bootsector+15),a       ;will make bytes checksum to 3 mod 256

        ld      b,0                     ;page 0 at 0000h
        ld      c,0                     ;sector 0
        ld      d,0                     ;track 0
        ld      e,0                     ;sector 1 (0 because of logical/physical trans.)
        ld      hl,bootsector           ;address of info. to write as boot sector
        ld      iy,dos_write_sector
        call    dodos                   ;actually write sector to disk
        pop     iy                      ;put iy back to 245.; can reference its sysled variables
        ld      sp,(oldstack)           ;put original stack back
        ret                             ;return to USR call in BASIC

dodos:
;
; IY holds the address of the DOS routine to be run. All other registers are passed
; intact to the DOS routine and are returned from it.
;
; Stack is somewhere in central 32K (conforming to DOS requirements), so saved AF and
; BC will not be switched out.
;
        push    af
        push    bc                      ;keep save registers while switching
        ld      a,(bankm)               ;RAM/ROM switching system variable
        or      7                       ;want RAM page 7
        res     4,a                     ;and DOS ROM
        ld      bc,bank1                ;port used for horiz ROM switch and RAM paging
        di
        ld      (bankm),a               ;keep system variables up to date
        out     (c),a                   ;RAM page 7 to top and DOS ROM
        ei
        pop     bc
        pop     af

        call    jumpto+                 ;go sub routine address in iy

        push    af                      ;return from DOS (if it will ever do so)
        push    bc
        ld      a,(bankm)
        and     0f6h                    ;reset bits for page 0
        set     4,a                     ;ROM to ROM 0 (48 BASIC)
        ld      bc,bank1
        di
        ld      (bankm),a
        out     (c),a                   ;switch back to RAM page 0 and 48 BASIC
        ei
        pop     bc
        pop     af
        ret

jumpto:
        jp      (iy)                    ;standard way to CALL (IY), by calling this jump

oldstack:
        dw      0                       ;somewhere to put BASIC's stack pointer
        ds      100
```

```
mystack:                        ;enough stack to meet +3DOS requirements

bootsector:

        .dephase                ;these are M80 pseudo ops. your assembler
        .phase  0FE00-          [illegible]


;
; Bootstrap will load into page 3 at address FE00h. The code will be entered at FE1Bh.
;
; Before it is written to track 0, sector 1, the bootstrap has byte 15 changed so
; that it will checksum to 3 mod 256.
;
; Boot will switch the memory so that the +3 BASIC ROM is at the bottom. Next up is
; page 5 - the screen, then page 2, and the top will keep page 3, as it would be
; unwise to switch out the bootstrap. BASIC routines can be called - to any RAM
; page switched in at the top, but the stack shouldn't be in the ISTACK area.
;

bootstart:
;
; The bootstrap sector contains the 16 byte disc specification at the start. The
; following values are for a AMSTRAD PCW range EP2/Spectrum +3 format disk.
;
        db      0                       ;+3 format
        db      0                       ;single sided
        db      40                      ;40 tracks [illegible] size
        db      9                       ;sectors per track

        db                              ;sg265127-7 - [illegible] error -
        db                              ;reserved track
        db      1                       ;blocks
        db                              ;directory blocks

        db      [illegible]             [illegible]
        db      [illegible]             [illegible]
        db      [illegible]             [illegible]


checksum:       db      2               ;checksum must = 3 mod 256 for this sec
;
; The bootstrap will be entered here - it has 4, 5, [illegible] 5 RAM pages in. I need to
; do print something, so I need +3 BASIC in at the bottom, page 5 (the screen) and
; system variables) next up. The next page - it be 2, and the top will be kept
; as page 3 because it still contains the bootstrap and stack (stack [illegible]
; on entry).
;
        di
        ld      a,(bank4)
        and     0F8h
        or      5                       ;RAM page 5 (as it holds bootstrap)
        set     4,a                     ;+3DOS and ROM
        ld      bc,bank4
        ld      (bank4),a
        [illegible]                     ;switch RAM and horizontal ROM
        [illegible]
        [illegible]
        0F8
        [illegible]                     [illegible] set bit 2 and reset bit [illegible] gives ROM [illegible]
        [illegible]
        [illegible]
        [illegible]                     [illegible] route row base 93,5

        ld      d,2
        call    setscr                  ;+3BASIC ROM routine to [illegible] screen 1A
        [illegible]
        call    print                   ;go [illegible] a message
```

```
eloop:
;
; end with an endless loop changing the border. This is where your own code
; for a game or operating system would go.
;
        ld      a,r                     ;a not-very-random random number
        out     (0feh),a                ;switch the border
        jr      eloop                   ;and loop

print:
        ld      a,(hl)                  ;this just loops printing characters
        cp      0ffh                    ;until it finds ffh
        ret     z
        rst     10h                     ;with 48K ROM in, this will print char in A
        inc     hl
        jr      print

message:
        db      16,2,17,7,19,0,22,10,13,"Hello, it's evening and welcome",0dh
diff:
        ds      512-(diff-bootstart),0  ;fill to end of sector with 0s

        end
```

There are one or two things that may be worth noting about this example. The first is that because BASIC normally has the address of the ERR NR system variable held in IY (so it can easily reference its system variables). It is important to store IY and replace it before returning to the original U S R call.

Just as before, the stack is moved so that it sits in the central 32K of memory. This will allow +3DOS routines to be called without having to move it again.

The dodos subroutine may be useful in your own programs. It only uses the IY register - which isn't used by the +3DOS system and allows a call to be made to any of the +3DOS routines.

The program uses DOS REF XDPB to make IX point at the relevant XDPB for drive A. It then logs in the disk in A so that it can be written to. After calculating and modifying the checksum byte for the information to be written to the boot sector of the disk, it writes the boot sector using DD WRITE SECTOR.

No checks are made to see that there is even a disk interface, and possible errors are ignored - the routine isn't designed to be used by those unfamiliar with possible pitfalls. The routine can be called with the BASIC command:

```
USR 28672
```

which will come back with whatever number BC happens to contain after completion of the routine.

The boot sector that is written to the disk has a standard disk specification in the first 16 bytes. This is followed by the bootstrap code that will be entered at address FE10h. As will be described in the interface for DOS BOOT (see part 27 of this chapter), the memory will initially be set up as 4, 7, 6, 3, however, the BASIC system variables are still intact and BASIC can be operated by switching in the correct ROM (3) to the bottom of memory and making sure that page 5 is in the 4000h - 7FFFh area of memory.

This very simple boot program just uses the BASIC ROM to print a greeting then enters a tight loop changing the border colour. It could be modified to load a large binary file and enter it or perform any other action you desired.

Part 27
Guide to +3DOS

Subjects covered...

- ROMs
- +3DOS interface
- File attributes and headers
- Disk format and specification
- Tracks and sectors
- Disk parameter blocks
- CP/M file compatibility
- Changing disks
- Logical to physical drive mapping
- +3DOS messages and requirements
- +3DOS routines

This section describes +3DOS - the disk operating system of the *+3*. The information will probably be of most interest to people familiar with assembly language [illegible] machine code) programming (see part 26 of this chapter for more information on this subject). What follows is highly technical and should not be used by the uninitiated.

The operating software of the *+3* is in effect held in *four* ROMs (though the [illegible] mation is actually contained in just two ICs). All four ROMs are addressed between 0000h and 3FFFh, although only one is switched in at a time.

ROM 0 is the 'editor' ROM and is the one entered when the *+3* is first switched on. This controls the high level 'menuing' and editing functions.

ROM 1 is the 'syntax' ROM and handles the high level control of *+3* BASIC. It contains the code for the BASIC parts of most of the disk based commands.

ROM 3 is the '48 BASIC' ROM and is virtually identical to the ROM used in the very first Spectrum. The only real area where it is different [illegible] in the code executed w[illegible] an interrupt occurs. If non zero a 'ticker' variable is decremented every second interrupt a[illegible] when it reaches zero the disk motor is switched off. This variable is held [illegible] page 7 along with some of the editor and DOS [illegible]ables. Page 7 will only be switched in (and this variable decremented) if bit 4 in the FLA[illegible] [illegible]iable is set - this is used by the software to identi[illegible] whether it is running 48 BASIC or *+3* B[illegible] 48 BASIC is selected (from the main menu or by the **SPECTRUM** com[illegible] this page-switching and ticker-decrementing won't happen. However [illegible] variable is subsequently set by your own program, this process will start [illegible] is still selected.

The keypad scanning routine of the Spectrum 128 and *+2* have been removed from ROM 3 in the *+3*.

A bug in the original 48 BASIC ROM has been fixed in the *+3*. When a non maskable interrupt (NMI) occurs, a jump is made to location 56h. This now checks the contents of the NMIADD system variable. If it is zero, a RETN is executed, otherwise a jump is made to the routine address held in NMIADD. The NMI code in ROM 2 consists of just a RETN.

ROM 3 not only provides the 48 BASIC mode for program compatibility, but executes the majority of *+3* BASIC commands that don't make use of the more advanced hardware of the *+3*.

The fourth ROM (ROM 2) holds +3DOS - the disk operating system. This is the subject of this section. Unlike the other ROMs, which are unlikely to be of much use for assembler programmers (except the 48 BASIC ROM perhaps), the +3DOS ROM has a wealth of routines that may well be of use in your own programs. We strongly recommend that any software that uses the disk drives makes use of these routines as they provide most of the facilities that one could wish for (more than are currently used by BASIC, in fact). Furthermore, the routines should only be accessed via the jump block. This not only makes it easier to write software that can be adapted to and from the AMSTRAD CPC range of computers, but also affords upwards compatibility for the future. The entry points for each routine are held in a jump table at address 0100h (256) in the ROM. Part 26 of this chapter gave a couple of examples of the way in which these routines can be called.

+3DOS provides the following facilities:

* Support for one or two floppy disk drives and a RAMdisk
* CP/M Plus and CP/M 3.2 file compatibility
* AMSTRAD CPC range and PCW range file and media compatibility
* Up to 16 files open at the same time
* Reading and writing files to or from any page in memory
* Byte level random access
* Deleting disk files, renaming disk files, changing disk files attributes
* Selecting the default drive and user
* Booting a game or operating system
* Low level access to floppy disk driver
* Optional mapping of two logical drives (A: or B:) onto one physical drive (unit 0).

+3DOS interface

+3DOS's interface is a set of routines accessed via a jump block. The routines provided fall into three categories:

* Essential firmware routines
* Additional routines for games and operating systems
* Low level floppy disk access routines for disk formatting, copying, etc.

The following is a list of the routines in each. These [illegible] routines of the routines function as

Essential filing system routines

NAME OF ROUTINE	FUNCTION
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

Additional routines for games and operating systems

NAME OF ROUTINE	FUNCTION
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

Low level floppy disk driving routines

NAME OF ROUTINE	FUNCTION
DD INTERFACE	Is the floppy disk driver interface present?
DD INIT	[illegible] disk driver
DD SETUP	[illegible] drive parameters
DD [illegible] RETRY	[illegible]
DD READ SECTOR	[illegible]
DD WRITE SECTOR	[illegible]
DD CHECK SECTOR	[illegible]
DD FORMAT	Format [illegible] track
DD READ ID	Read a [illegible]
DD TEST UNSUITABLE	[illegible]
DD [illegible]	[illegible]
DD SET FORMAT	[illegible] XDPB1 [illegible] DD FORMAT
DD ASK	[illegible] (extcmd [illegible] drive present
DD DRIVE STATUS	Fetch drive [illegible]
DD EQUIPMENT	What type [illegible]
[illegible] NC[illegible]	[illegible]
[illegible]	[illegible]
DD L[illegible]	[illegible]
DD L SEEK	[illegible]
DD L R[illegible]	[illegible]
DD L WRITE	[illegible]
DD L ON MOTOR	[illegible]
DD L OFF MOTOR	[illegible]
DD L OFF [illegible] TOR	[illegible]

Games and other non-BASIC programs

+3D [illegible] rovide [illegible]

- [illegible] hole machine uses the [illegible]
- [illegible] some [illegible] program to take control [illegible] of required [illegible] off and [illegible] prevent any bank [illegible]
- [illegible] without [illegible]

Using +3DOS without a floppy disk interface

Even if the floppy disk interface were not present, +3DOS could still be used as follows

* Only drive M is available (the RAMdisk)
* The default drive for filenames is initialised to M rather than A
* Any attempt to use drives A or B will fail with error 22 - Drive not found
* As the sector cache is not required for use with RAMdisk, the size of the RAMdisk is increased to 64K (the whole of pages 1, 3, 4, 6). This will give 62K of data and 2K of directory (64 entries)
* The presence of the floppy disk interface can be determined by calling DD INTERFACE. If the interface were not present, then none of the other low level floppy disk routines (DD etc.) could be called, the effect of so doing is undefined

File attributes

Bit 7 of the name and type field characters are the file attributes. The top bits of the name field characters are denoted f1 - f8. The top bits of the type field characters are denoted f1 - f3. They have the following meanings

f1 - f4	Available to the user
f5 - f8	Reserved (always 0)
f1	0 means file is read-write, 1 means file is read-only
f2	0 means not system file, 1 means system file
f3	0 means not archived, 1 means archived

A read-only file cannot be written to, erased or renamed. System files can optionally be omitted from the directory catalog. The archive attribute is ignored by +3DOS

Newly created files have all attributes set to 0. An existing file's attributes can only be changed by DOS SET ATTRIBUTES (as used by BASIC's **MOVE** command)

File headers

Tape files have headers which contain some sy[illegible] headers. All files created by BASIC's **SAVE** [illegible]

The +3DOS header mechanism provides a dedicated [illegible] byte area [illegible] BASIC's use. The remainder of the hea[illegible] ved for +3DOS [illegible] created by BASIC commands (see DOS OPEN description)

+3DOS files may have a single header in the first 128 bytes of the file - the ***header record***. These headers are detected by a 'signature' and checksum. If the signature and checksum are as expected, then a header is present; if not, then there is no header. Thus, it is possible, but unlikely, that a file without a header could be mistaken for one with a header.

The format of the header record is as follows:

Bytes 0 - 7	+3DOS signature - 'PLUS3DOS'
Byte 8	- 1A (26) Soft-EOF (end of file)
Byte 9	- Issue number
Byte 10	Version number
Bytes 11 - 14	- Length of the file in bytes. 32 bit number, least significant byte in lowest address
Bytes 15 - 22	+3 BASIC header data
Bytes 23 - 126	- Reserved (set to 0)
Byte 127	- Checksum (sum of bytes 0 - 126 modulo 256)

The issue and version numbers are provided for any future expansion. The issue number must equal the software's issue number, the version number must be less than or equal to the software's version number.

+3DOS performs all the necessary header 'house-keeping'. A pointer to +3 BASIC's 8 byte header area may be returned using DOS REF HEAD. It is never necessary to write directly to the 128 byte header.

AMSDOS headers (as used on the AMSTRAD CPC range of computers) will not be recognised. AMSDOS files will be treated by +3DOS as headerless and vice versa.

Disk formats

+3DOS supports exactly the same disk format as CP/M Plus and LocoScript on the AMSTRAD PCW range of computer word processors (ie. the first format listed below).

The following formats are automatically detected when the disk is first accessed:

* AMSTRAD PCW range single track (eg. as used on model PCW8256)
* AMSTRAD PCW range double track (eg. as used on model PCW8512)
* AMSTRAD CPC range system format
* AMSTRAD CPC range vendor format
* AMSTRAD CPC range data only format

Note that the AMSTRAD CPC range's IBM format is ***not*** supported.

Other disk formats can be used by patching the XDPB for a drive. The XDPB is the same as for the first format listed above. It is ***not*** the same as on the CPC range.

Disk formats are subject to the following restrictions

* 512 byte sector size
* Maximum of 255 sectors per track
* Maximum of 255 tracks
* Maximum of 256 directory entries
* Maximum of 360 allocation units

Logical tracks and sectors

The disk driver routines require logical tracks and sectors. These are used to hide information concerning the number of sides and the actual sector numbers from +3DOS, which knows nothing about them.

Logical track numbers on a single sided disk are the same as physical track numbers.

For double sided disks, two options are available.

1. Alternating sides

```
side 0 track 0 = logical track 0
side 1 track 0 = logical track 1
side 0 track 1 = logical track 2
side 1 track 1 = logical track 3
  to
side 0 last track = logical track n-2
side 1 last track = logical track n-1
```

2. Successive sides

```
side 0 track 0 = logical track 0
side 0 track 1 = logical track 1
side 0 track 2 = logical track 2
  to
side 0 last track = logical track n/2-1
  and then
side 1 last track-1 = logical track n/2
side 1 last track-2 = logical track n/2+1
side 1 last track-3 = logical track n/2+2
  to
side 1 track 0 = logical track n-1
```

where n is the total number of logical tracks (ie. 2 x number of tracks per side).

Logical sectors hide the actual physical sector numbers. Logical sector numbers always start from 0.

```
Logical sector = physical sector - first sector
```

Disk specification

The PCW range disk [illegible] *+3* [illegible] which is defined in the [illegible] for 1 track [illegible] The format used on the *+3* [illegible] and the specification is also that [illegible] in the second example [illegible]

Byte	Description
	= [illegible] PCW range [illegible] *+3*
	= [illegible] CPC [illegible] DSSSI system format
	= [illegible] CPC [illegible] SSI data only format
	= [illegible]
	[illegible]
[illegible]	[illegible]
	= [illegible]
	= [illegible]
	= [illegible]
	[illegible]
	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

[illegible]

Extended disk parameter blocks (XDPB)

[illegible] This [illegible]

[illegible]

Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Bytes [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Bytes [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
	= [illegible]
	= [illegible]
	= [illegible]
	[illegible]
	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Bytes [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
Byte [illegible]	[illegible]
	= [illegible]
	= [illegible]
	[illegible]
	= [illegible]
	= [illegible]

[illegible]	[illegible]

[illegible]

AMSTRAD PCW range single track format (type 0) (As used by the +3)

[illegible]	[illegible]

AMSTRAD CPC range SYSTEM format (type 1)

AMSTRAD CPC range DATA ONLY format (type 2)

[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

CP/M File compatibility

+3DOS uses the CP/M file structure subject to the following restrictions:

- Maximum file size of [illegible] megabyte (CP/M 2.2 supports a maximum of [illegible] megabytes)
- [illegible] CP/M Plus supports a maximum of 128 megabytes)
- [illegible]
- [illegible] erased, renamed, etc. along with their file(s) but do [illegible]
- [illegible] when a file is created [illegible] otherwise to [illegible]
- [illegible] OS_SE1 [illegible]

File model

[illegible] associated with [illegible] is the [illegible] to be written/read. The [illegible] pointer is automatically [illegible] the user may set it [illegible] for random [illegible]

The end of [illegible] EOF [illegible] next byte [illegible] Files without header [illegible] to the [illegible] ceiling (EOF) [illegible]

[illegible]

[illegible]

[illegible]

Changing disks

Under +3DOS [illegible] there are no [illegible]

A disk should n[illegible] is [illegible] then [illegible] + [illegible] the correct [illegible] + [illegible]

Note that cha[illegible]

Logical to physical drive mapping

[illegible]

[illegible] required [illegible] HAV[illegible] IGI[illegible] passed [illegible] message on the [illegible]

```
Please put the disk for : into
the drive then press any key
```

(where [illegible] is the [illegible] of the [illegible] A: or B: [illegible] wait for a key to be pr[illegible] Note [illegible] required [illegible] in

[illegible] SMART [illegible] to isec [illegible] Tumu [illegible] level then drive B is disabled

+3DOS Error codes

Many +3DO[illegible] [illegible] with drive B [illegible] The error [illegible]

Recoverable [illegible] rors

	Drive not ready
	Disk is write protected
	Seek fail
	CRC data error
:	No data
	Missing address mark
	Unrecognised disk format
	Unknown disk error
-	Disk changed whilst +3DOS was using it
	Unsuitable media for drive

Non-recoverable errors

[illegible]	Bad filename
[illegible]	Bad parameter
[illegible]	Drive not found
[illegible]	File not found
[illegible]	File already exists
[illegible]	End of file
[illegible]	Disk full
[illegible]	Directory full
[illegible]	Read-only file
[illegible]	File number not open [illegible] or open with [illegible]
[illegible]	Access denied [illegible] already)
[illegible]	Cannot rename between drives
[illegible]	Extent missing [illegible] should be there
[illegible]	Uncached [illegible]
[illegible]	File too big [illegible] trying to [illegible]
[illegible]	Disk not bootable [illegible]
[illegible]	Drive in use [illegible]

A [illegible] Unsuitable media for drive [illegible] used by trying to write [illegible] a double track drive, or trying to read or write a double track disk in a single track drive.

The [illegible] Missing address mark [illegible] disk that is not formatted (although [illegible]

+3DOS Messages

[illegible] the [illegible] in the event of a recoverable disk error [illegible] prompted to - Retry, Ignore or Cancel? [illegible] R [illegible] the reply is I then the error is ignored [illegible] C [illegible] and an error condition is [illegible] recoverable then [illegible]

The recoverable disk errors [illegible]

[illegible]	Drive x: not ready
[illegible]	Drive x: disk write protected
[illegible]	Drive x: track ##, seek fail
[illegible]	Drive x: track ##, sector ss, data error
[illegible]	Drive x: track ##, sector ss, no data
[illegible]	Drive x: track ##, sector ss, missing address mark
[illegible]	Drive x: bad format
[illegible]	Drive x: track ##, sector ss, unknown error
[illegible]	Drive x: disk changed, please replace
[illegible]	Drive x: disk unsuitable

where x is the disk drive (eg A: or B: ## is the track [illegible] the sector number

The above messages are [illegible] followed by - Retry, Ignore or Cancel?

The ALERT routine is [illegible] messages if the error occurs [illegible] committed to execute a DOS routine [illegible] if DOS OPEN is called (with access [illegible] write or read write) and the disk in the drive [illegible] carry clear and A= [illegible] the ALERT [illegible]

If however while reading data during DOS READ a bad sector is found ALERT will be called to warn the user. This will then offer the opportunity of retry (eg if for [illegible] the disk was not properly seated in the drive) ignoring (so that the bad sector will be ignored allowing as much of the file as possible to be recovered) or cancelling (perhaps because the problem is obviously irrecoverable)

(Note that the routine interface for DOS SET MESSAGE has changed between [illegible] V1.0 and V1.1 of +3DOS. It is important therefore that DOS VERSION is called and that if the vers[illegible] mark are greater than V1.0 the new routine [illegible] used. This is the only char[illegible] 1.0 and V1.1 [illegible]

+3DOS requirements

When any of the +3DOS routines are called the [illegible]

C000h FFFFh (49[illegible]
8000h BFFFh ([illegible]
4000h 7FFFh ([illegible]
0000h 3FFFh (0 [illegible] 16383)

The stack must be below BFE0h (49120) [illegible] above [illegible]0h (16384). The upper [illegible] BF00h (rather than C000h) because the top 30 b[illegible] used to implement [illegible]ter-page blo[illegible] moves. This [illegible] have at least 50 words available.

+3DOS supports up to 16 file [illegible] are utilised by +3 BASIC so it would be u[illegible] -3 [illegible]IC command might be executed while a file is st[illegible] ns an error (even if the report is 0 OK).

For each of the routines described in this section interrupts must be enabled on entry and will still be enabled on exit.

+3DOS Store usage

RAM [illegible]

[illegible]

All +3DOS [illegible]

The [illegible] which they [illegible]

The [illegible] ROM [illegible] routine [illegible]

Essential filing system routines

DOS INITIALISE
0100h (256)

[illegible]

ENTRY CONDITIONS

[illegible]

EXIT CONDITIONS

[illegible]

DOS VERSION
0103h (259)

Get the DOS issue and version numbers

ENTRY CONDITIONS

None

EXIT CONDITIONS

D = Issue
E = Version (within issue)

Always

AF BC HL IX corrupt
All other registers preserved

DOS OPEN
0106h (262)

Create and/or open a file

There is a choice of action depending on whether or not the file already exists. The choices are open action (or create action) and are specified in DE. If the file already exists then the open action is followed, otherwise the create action is followed.

Open action

1. Error - File already exists
2. Open the file [illegible] the header (if any). Position file pointer after header
3. Open the file [illegible] any header. Position file pointer at 0000[illegible]
4. Assume given filename is *filename*.*type*. Erase *filename*.BAK (if it exists). Rename *filename*.*type* to *filename*.BAK. F[illegible]
5. Erase existing version. Follow create act[illegible]

Create action

1. Error - File does not exist
2. Create and open new file with a header. Position file pointer [illegible]
3. Create and open new file without a header. Position file pointer at 00[illegible]

[illegible] To simulate the *tape* action [illegible] the file exists open it otherwise create it with a header set open action = [illegible] create action =

To [illegible] if the file does not exist set open action = [illegible] create action = [illegible]

To [illegible] with header first rename any existing version to .BAK set open action = [illegible] create action =

Files with header [illegible] the smallest byte position greater than all written byte positions

Files without header have their [illegible] record [illegible] as the byte at the start of the smallest 128 byte record [illegible] greater than all written record positions

Soft-EOF is the character (Ah 26) and is nothing to do with the EOF position only the routine DOS BYTE READ [illegible] is about soft-EOF

The header data [illegible] be used by the [illegible] any purpose whatsoever If open action = [illegible] has a header then the header data is read from the file otherwise the header [illegible] The header data is available even if the file does not have a header Call DOS REF HEADER [illegible] the header data

Note that **+3** BASIC [illegible] of the first 7 of the [illegible]

BYTE	0	1	2	3	4	5	6
Program	[illegible]	[illegible]	[illegible]	[illegible]	LINE	[illegible]	to prog
Numeric array	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Character array	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
CODE SCREENS	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

[illegible] = [illegible]

[illegible] LOAD [illegible] should be filled with [illegible]

[illegible] has a header [illegible]

A [illegible] may be opened for [illegible]

A [illegible] with [illegible] 3 write [illegible]

ENTRY CONDITIONS

B = File number (0 - 15)
C = Access mode required
Bits 0, 2 values
1 = exclusive-read
2 = exclusive-write
3 = exclusive-read-write
5 = shared-read
Bits 3, 7 = (reserved)
D = Create action
E = Open action
HL = Address of [illegible] (no wildcards)

EXIT CONDITIONS

If file newly created
Carry true
Zero true
A corrupt
If existing file [illegible]
Carry true
Zero false
A corrupt
Otherwise
[illegible]
A = [illegible]
Always
BC DE HL IX corrupt
All other registers preserved

DOS CLOSE
0109h (265)

Close a file

Write the header (if there [illegible]

Write any outstanding [illegible]

Update the directory

Release the file number

All opened files must eventually be closed [illegible] (A [illegible] cannot be reused until it is closed (or abandoned))

ENTRY CONDITIONS

B = File number

EXIT CONDITIONS

If OK

Carry [illegible]

A [illegible]

Otherwise

[illegible]

A = [illegible]

Always

[illegible]

A [illegible]

DOS ABANDON
010Ch (268)

Abandon [illegible]

Similar [illegible] DOS CLOSE [illegible] header [illegible] directory [illegible] yet to be written to disc [illegible] used [illegible] force [illegible] sed in the event that DOS CL[illegible] [illegible] if the media is [illegible] [illegible]ently changed or removed)

ENTRY CONDITIONS

B = File number

EXIT CONDITIONS

If OK

Carry [illegible]

A corrupt

[illegible]

[illegible]

A = [illegible]

Always

[illegible]

A [illegible]

DOS REF HEAD
010Fh (271)

Point at the hea[illegible]

The [illegible] It is [illegible] with [illegible]

Note that **+3** BASIC uses these 8 bytes (see the note under DOS OPEN which gives the details). If creating a file that will subsequently be **LOAD**ed within BASIC, then these bytes should be filled with the relevant values.

ENTRY CONDITIONS

B = File number

EXIT CONDITIONS

If OK, but file doesn't have a header:
- Carry true
- Zero true
- A corrupt
- IX = Address of header data in page 7

If OK, file has a header:
- Carry true
- Zero false
- A corrupt
- IX = Address of header data in page 7

Otherwise:
- Carry false
- A = Error code
- IX corrupt

Always:
- BC DE HL corrupt
- All other registers pr[illegible]

DOS READ
0112h (274)

Read bytes from a file [illegible] memory.

Advance the file pointer.

The destination buffer is [illegible] the fo[illegible] memory [illegible]

C000h	FFFFh	[illegible]
8000h	BFFFh	[illegible]
4000h	7FFFh	[illegible]
0000h	3FFFh	[illegible]

This routine does [illegible]

Reading EOF [illegible]

ENTRY CONDITIONS

B = File number

C = [illegible]

DE = Number of bytes to [illegible]

HL = Address for bytes to be read

EXIT CONDITIONS

[illegible]

Carry flag

A = [illegible]

Other [illegible]

[illegible]

A = [illegible]

DE = Number of bytes remaining [illegible]

A = [illegible]

BC DE HL IX corrupt

All other registers [illegible]

DOS WRITE
0115h (277)

Write bytes to a file from memory.

Advance the file pointer.

The source address is in the following memory configuration:

C000h	FFFFh	(49[illegible]
8000h	BFFFh	[illegible] -915[illegible]) - Page 2
4000h	7FFFh	[illegible] 32767) [illegible]
0000h	3FFFh	(0 - 16383) [illegible]

ENTRY CONDITIONS

B = [illegible]

C = [illegible]

DE = Number of bytes [illegible]

HL = Address [illegible]

EXIT CONDITIONS

[illegible]

Carry flag

A = [illegible]

Other [illegible]

Carry flag

A = [illegible]

DE = Number of bytes [illegible]

A corrupt

BC HL IX corrupt

All other registers [illegible]

DOS BYTE READ
0118h (280)

Read a byte from a file

Advance the file pointer

Tests for soft EOF (1Ah (26)). As this condition is not latched, it is possible to read past soft-EOF

EOF is latched

The caller must decide whether or not soft-EOF is of interest. This would normally be the [illegible] when processing an ASCII file

Reading EOF will produce an error

ENTRY CONDITIONS

B = File number

EXIT CONDITIONS

If OK, Byte <> 1Ah (26), soft-EOF
- Carry true
- Zero false
- A corrupt
- C = Byte

If OK, Byte = 1Ah (26), [illegible]
- Carry true
- Zero true
- A [illegible]
- C = Byte

[illegible]
- Carry [illegible]
- A = [illegible]
- C corrupt

Always
- B DE HL IX c[illegible]
- All other registers [illegible]

DOS BYTE WRITE
011Bh (283)

Write a byte to a file

Advance the file pointer

ENTRY CONDITIONS

B = File number
C = Byte to write

EXIT CONDITIONS

If OK

Carry true

A corrupt

Otherwise

[illegible]

Always

BC' DE' HL' IX corrupt

All other registers preserved

DOS CATALOG
011Eh (286)

Fills a buffer with part of the directory (sorted)

The filename specifies the drive, user and a (possibly ambiguous) filename

Since the size of a directory is variable (and may be quite large), this routine permits the directory to be catalogued in a number of small sections. The caller passes a buffer pre-loaded with the first required filename (or zeros for the start of the directory). The buffer is loaded with part (or all, if it fits) of the directory sorted in ASCII order. If more of the directory is required, this routine is re-called with the buffer re-initialised with the last file previously returned. This procedure is followed repeatedly until all of the directory has been [illegible]

Note that +3DOS format disks (which are the same as single sided, single track AMSTRAD PCW range format disks) may have a maximum of 64 directory entries

Buffer format

[illegible]

[illegible] ded with the first *filename.type* required. Entry [illegible] will contain the first [illegible] ter than the preloaded entry [illegible] returned preload entry is OK

[illegible] r the directory, this [illegible] with entry 0 replaced by [illegible] xt part of the directory

Entry format (13 bytes long)

Bytes [illegible]

[illegible]

[illegible]

ENTRY CONDITIONS

B = n + 1, Size of buffer in entries, >= 2
C = Filter
bit 3 = include system files (if set)
bits [illegible] = [illegible]
DE = Address of buffer (first entry initialised)
HL = Address of [illegible] which [illegible] entries

EXIT CONDITIONS

If OK
Carry true
A corrupt
B = Number of completed entries in buffer, 0...n (If B = n there may be more to come)
Otherwise
Carry false
A = Error code
B corrupt
Always
C DE HL IX corrupt
All other registers preserved

DOS FREE SPACE
0121h (289)

How much free space is there on this drive?

ENTRY CONDITIONS

A = Drive, ASCII A ... P

EXIT CONDITIONS

If OK
Carry true
A corrupt
HL = Free space (in kilobytes)
Otherwise
Carry false
A = Error code
HL corrupt
Always
BC DE IX corrupt
All other registers preserved

DOS DELETE
0124h (292)

[illegible]

[illegible]

ENTRY CONDITIONS

[illegible]

EXIT CONDITIONS

[illegible]

DOS RENAME
0127h (295)

[illegible]

[illegible]

ENTRY CONDITIONS

[illegible]

EXIT CONDITIONS

[illegible]

DOS BOOT
012Ah (298)

[illegible]

[illegible]

[illegible]

[illegible]

ENTRY CONDITIONS
[illegible]

EXIT CONDITIONS
[illegible]

DOS SET DRIVE
012Dh (301)

[illegible]

[illegible]

[illegible]

[illegible]

ENTRY CONDITIONS
[illegible]

EXIT CONDITIONS

If OK

Carry true

A = [illegible]

Otherwise

Carry false

A = [illegible]

Always

[illegible]

[illegible]

DOS SET USER
0130h (304)

Set the default user [illegible] user number

The default user [illegible]

This only a[illegible] parameter

ENTRY CONDITIONS

A = User [illegible] FFh [illegible]

EXIT CONDITIONS

If OK

Carry true

A = [illegible]

Otherwise

Carry false

A = [illegible]

Always

[illegible]

[illegible]

Additional routines for games and operating systems

DOS GET POSITION
0133h (307)

Get the file pointer

ENTRY CONDITIONS

B = File number

EXIT CONDITIONS

If OK

Carry true

A corrupt

E HL = File pointer 000000h - FFFFFFh (0 - 16777215)

(E holds most significant byte, L holds least significant byte)

Otherwise

Carry false

A = Error code

E HL corrupt

Always

BC D IX corrupt

All other registers preserved

DOS SET POSITION
0136h (310)

Set the file pointer

Does not access the disk

Does not check (or care) if pointer is > = 3 megabytes

ENTRY CONDITIONS

B = File number

E HL = File pointer 000000h - FFFFFFh (0 - [illegible]

(E holds most significant byte, L holds least significant byte)

EXIT CONDITIONS

If OK

Carry true

A corrupt

Otherwise

Carry false

A = Error code

Always

BC DE HL IX corrupt

All other registers preserved

DOS GET EOF
0139h (313)

Get the end of file (EOF) file position, i.e. the lowest byte position greater than all written byte positions

Does not affect the file pointer

Does not consider soft-EOF

ENTRY CONDITIONS

B = File number

EXIT CONDITIONS

If OK

Carry true

A corrupt

E:HL = File pointer 000000h - FFFFFFh (0 - 16777215)

(E holds most significant byte, L holds least significant byte)

Otherwise

Carry false

A = Error code

E:HL corrupt

Always

BC D IX corrupt

All other registers preserved

DOS GET 1346
013Ch (316)

Get the current location of the cache and RAMdisk

Pages 1, 3, 4, 6 are considered as an array of 128 sector buffers (numbered 0 - 127), each of 512 bytes. The cache and RAMdisk (which are two separate contiguous) are part of this array

Any unused sector buffers may be used by the caller

Note that the sizes may be smaller than those specified in DOS SET 1346 as there is a (unpublished) maximum size of cache and a minimum size of RAMdisk (4 sectors)

ENTRY CONDITIONS

None

EXIT CONDITIONS

L = First buffer in cache

D = Number of cache sector buffers

H = First buffer for RAMdisk

E = Number of RAMdisk sector buffers

Always

AF BC IX corrupt

All other registers preserved

DOS SET 1346
013Fh (319)

Rebuild the sector cache and RAMdisk

This routine is used to make some store available to the user or to return store to DOS

Note that if the RAMdisk is moved or its size is changed then all files thereon are erased

Pages 1, 3, 4, 5 are considered as an array of 128 sector buffers numbered 0 (127) each of 512 bytes. The cache and RAMdisk occupy two separate (contiguous) areas of this array

The location and size of the cache and RAMdisk can be specified separately, any remaining buffers are unused by DOS and are available to the caller

The cache and RAMdisk must not overlap, +3DOS does not check this, responsibility lies with the caller

Note that the sizes actually used may be smaller than those specified as in practice there is a maximum cache size and a minimum size of RAMdisk (4 sectors)

A cache size of 0 will still work but will seriously impair the floppy disk performance

This routine will fail if there are any files open on drive M

ENTRY CONDITIONS

D = First buffer for cache
E = Number of cache sector buffers
H = First buffer for RAMdisk
L = Number of RAMdisk sector buffers
(Note that D + E <= 128)

EXIT CONDITIONS

If OK
Carry true
A corrupt
Otherwise
Carry false
A = Error code
Always
BC DE HL IX corrupt
All other registers preserved

DOS FLUSH
0142h (322)

Write any pending headers and directory entries for this drive

This routine ensures that the disk is up to date. It [illegible] called at any time [illegible]

ENTRY CONDITIONS

A = Drive, ASCII A [illegible]

EXIT CONDITIONS

[illegible]

DOS SET ACCESS
0145h (325)

[illegible]

ENTRY CONDITIONS

[illegible]

EXIT CONDITIONS

[illegible]

DOS SET ATTRIBUTES
0148h (328)

Set a file's attributes

Only the file attributes [illegible] [4], [1] [illegible] [4] can be set or cleared. The [illegible] attributes [5]...[illegible] always 0

This routine first sets the attributes specified in D, then [illegible] attributes specified in E. [illegible] E has priority

ENTRY CONDITIONS

D = Attributes to set
- bit 0 = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]

E = Attributes [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]
- bit [illegible] = [illegible]

HL = Address [illegible] whose [illegible] as permitted

EXIT CONDITIONS

If OK
- Carry true
- A corrupt

Otherwise
- Carry [illegible]
- A = Error [illegible]

Always
- BC DE HL IX corrupt
- All other registers preserved

DOS OPEN DRIVE
014Bh (331)

Open the disk in this drive as a single file

The whole disk is presented [illegible] used to examine poke dire[illegible] [illegible] of by anyone who values their files

Sets file pointer to 000000h (0)

If there are any files open on this drive from other file numbers with shared-read access, then the disk can only be opened with shared-read access from this file number

If there are any files open on this drive from other file numbers with exclusive access, then the disk cannot be opened from this file number

ENTRY CONDITIONS

A = Drive, ASCII A P
B = File number
C = Access mode required
Bits 0 2 values
1 = exclusive-read
2 = exclusive-write
3 = exclusive-read-write
5 = shared-read
(all other bit settings reserved)
Bits 3, 7 = 0 (reserved)

EXIT CONDITIONS

If OK
Carry true
A corrupt
Otherwise
Carry false
A = Error code
Always
BC DE HL corrupt
All other registers preserved

DOS SET MESSAGE
014Eh (334)

Enable/disable disk error messages

This should be used to make +3DOS aware of your own ALERT subroutine. When +3DOS detects an error, it will call your ALERT subroutine, passing to it the values documented below. The ALERT subroutine should print the text of the message that +3DOS passes it, then should wait for the user to press a key. If the key is in the reply string (that +3DOS also passes - version V1 . only), then a *ret* should be made with A = 0 , 1 or 2, or containing the character (depending on the version of +3DOS)

ENTRY CONDITIONS

A = Enable/disable
FFh (255) = enable
00h (0) = disable
HL = Address of ALERT routine (if enabled)

EXIT CONDITIONS

HL = address of previous ALERT routine [illegible]

Always

AF BC DE IX [illegible]

All other registers preserved

NOTE

Note that if you [illegible] the conditions pa[illegible] subroutine must prod[illegible]

Note that there [illegible] +3DOS [illegible] V1 [illegible]

ALERT (VERSION V1.0 ONLY)

ENTRY CONDITIONS

DE = Addr[illegible]

HL = Address [illegible]

EXIT CONDITIONS

A = [illegible]

[illegible]

F BC DE HL IX [illegible]

All other register [illegible] reser[illegible]

The sec[illegible] general [illegible]

ALERT (VERSION V1.1 AND ABOVE)

ENTRY CONDITIONS

A = Error [illegible]

[illegible]

[illegible]

[illegible]

HL = A [illegible] (terminal entry [illegible]

EXIT CONDITIONS

A = [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

If you pr[illegible]oe [illegible]

DOS REF XDPB
0151h (337)

Point at the XDPB [illegible]

ENTRY CONDITIONS

A = [illegible] ASCII A[illegible]

EXIT CONDITIONS

[illegible]

DOS MAP B
0154h (340)

Map drive B = [illegible]

If [illegible] is [illegible] it isn't then [illegible] 3[illegible]

If mapping [illegible] then [illegible]

ENTRY CONDITIONS

[illegible] = Jol[illegible]

[illegible] = Addr[illegible]

EXIT CONDITIONS

[illegible]

NOTE

The definition of the subroutine CHANGE DISK is as shown ahead. Note that if you are substituting your own CHANGE DISK subroutine, the entry conditions are the conditions passed to your subroutine and the exit conditions are registers you are allowed to corrupt.

CHANGE DISK

Ask the user to change the disk in unit 0.

Wait for the user to acknowledge the change.

ENTRY CONDITIONS

A = Logical drive ASCII 'A' - 'P'
HL = Address of message (page 7) terminated by FFh (255)

EXIT CONDITIONS

Always
AF BC DE HL IX corrupt
All other registers preserved

Low level floppy disk driving routines

The following are the floppy disk driver routines. The unit number is 0...3 for the μPD765A. On the *+3* unit 0 is drive A, and unit 1 is drive B - or optionally both A and B may be mapped onto unit 1. Units 2 and 3 are not used.

With the exception of DD INTERFACE, none of these routines may be called if the floppy disk interface is not present.

All routines assume that interrupts are enabled on entry, and will still be enabled on exit.

DD INTERFACE
0157h (343)

Is the floppy disk drive interface present? (This information is also held by BASIC in bit 4 of the FLAGS3 system variable.)

ENTRY CONDITIONS

None

EXIT CONDITIONS

If present
Carry true
Otherwise
Carry false
Always
A BC DE HL IX corrupt
All other registers preserved

DD INIT
015Ah (346)

Initialise the disk driver

ENTRY CONDITIONS
None

EXIT CONDITIONS
Always
AF BC DE HL IX corrupt
All other registers preserved

DD SETUP
015Dh (349)

Set up disk parameters

Send a specify command

Parameter block format

Byte 0 - Motor on time (in 100 mS units)
Byte 1 - Motor off time (in 100 mS units)
Byte 2 - Write off time (in 10 µS units)
Byte 3 - Head settle time (in mS units)
Byte 4 - Step rate (in mS units)
Byte 5 - Head unload time (in [illegible] mS units 32 - 480)
Byte 6 - (Head load time x 2) + 1 (in 4 mS units 4 - 508)

ENTRY CONDITIONS
HL = Address of parameter block

EXIT CONDITIONS
Always
AF BC DE HL IX corrupt
All other registers preserved

DD SET RETRY
0160h (352)

Set the try and retry [illegible] A [illegible] (0 = no retry.)

ENTRY CONDITIONS
A = Try/retry count > =

EXIT CONDITIONS
Always
AF BC DE HL IX corrupt
All other registers [illegible]

DD READ SECTOR
0163h (355)

Read 1 sector

ENTRY CONDITIONS
B = Page for 2000h (49152) [illegible] FFFFh (6[illegible]
C = [illegible]
L = [illegible] base
E = [illegible] sector [illegible] base
HL = Address of buffer
IX = Addr[illegible] XDPB

EXIT CONDITIONS
If OK
Carry true
A corrupt
Otherwise
Carry [illegible]
A = Error code
Always
BC DE HL IX corrupt
All other registers [illegible]

DD WRITE SECTOR
0166h (358)

Write a sector

ENTRY CONDITIONS

B = Page for 2000h (49152) - FFFFh(65535)
C = Unit (0/1)
D = Logical track (0 base)
E = Logical sector (0 base)
HL = Address of buffer
IX = Address of XDPB

EXIT CONDITIONS

If OK
Carry true
A corrupt
Other case
Carry false
A = Error code
Always
BC DE HL IX corrupt
All other registers preserved

DD CHECK SECTOR
0169h (361)

Check sector [illegible] the µ[illegible]

Checks that the [illegible]

Note that FFh (255 [illegible] anything (see µPD765A specification for further details)

ENTRY CONDITIONS

B = Page for 2000h (49152) - FFFFh(65535)
C = Unit
D = [illegible]
E = [illegible]
HL = [illegible]
IX = [illegible]

EXIT CONDITIONS

If OK (equal)
- Carry true
- Zero true
- A corrupt

If OK (not equal)
- Carry true
- Zero false
- A corrupt

Otherwise
- Carry false
- A = Error code

Always
- BC DE HL IX corrupt
- All other registers preserved

DD FORMAT
016Ch (364)

Format a track. (Uses the μPD765A format track command.)

Buffer contains 4 bytes for each sector as follows

C	Track number	(0...39)
H	Head number	(always 0 on the +3's single sided drives)
R	Sector number	(0...255)
N	Log2(sector size)-7	(2 for 512 byte sectors)

ENTRY CONDITIONS

B = Page for C000h (49152) FFFFh(65535)
C = Unit (0/1)
D = Logical track 0 base
E = Filler byte, usually E5h (229)
HL = Address of format buffer
IX = Address of XDPB

EXIT CONDITIONS

If OK
- Carry true
- A corrupt

Otherwise
- Carry false
- A = Error code

Always
- BC DE HL IX corrupt
- All other registers preserved

DD READ ID
016Fh (367)

Read [illegible] identifier

ENTRY CONDITIONS

C = Unit (0/1)
D = Logica[illegible]
IX = Addr[illegible]

EXIT CONDITIONS

If OK
Carry true
A = [illegible]
Otherwise
Carry false
A = [illegible]
Always
HL = [illegible] (result buffer in case 7)
[illegible]
All other registers preserved

DD TEST UNSUITABLE
0172h (370)

Check that disk is suitable [illegible]

A single track disk will not [illegible] a double track drive and vice versa

ENTRY CONDITIONS

C = [illegible]
IX = [illegible]

EXIT CONDITIONS

If suitable
Carry true
A corrupt
Otherwise
Carry false
A = Error code
Always
BC DE HL IX corrupt
All other registers preserved

DD LOGIN
0175h (373)

Log in a new drive

Initialise the XDPB

This routine does not [illegible]

ENTRY CONDITIONS

C = Drive (0,1)
IX = Address [illegible]

EXIT CONDITIONS

If OK
Carry [illegible]
A = [illegible]
[illegible]
HL = [illegible]
Otherwise
[illegible]
A = Error [illegible]
DE HL [illegible]
A [illegible]
BC IX [illegible]
All other [illegible]

DD SEL FORMAT
0178h (376)

Initialise an XDPB [illegible] format

This routine does not affect [illegible]

ENTRY CONDITIONS

A = [illegible]
= [illegible] +3 [illegible]
= [illegible]
= [illegible]
= [illegible]
[illegible] = [illegible]
X = Address [illegible]

EXIT CONDITIONS

If OK

Carry true

A = Disk type

[illegible] = size of 2 bit [illegible]

[illegible] = size of hash table

[illegible]

Carry [illegible]

A = [illegible]

[illegible]

[illegible]

[illegible]

All other registers preserved

DD ASK 1
017Bh (379)

Check to see if unit [illegible] present (BASIC holds this information in bit [illegible] of the FLAGS3 system variable.)

Turn motor on

Fetch drive status

If unit [illegible] is not-ready and write-protected, then unit [illegible] is missing. Start motor off timeout

Note that this routine can be fooled by [illegible] are almost, but not quite, inserted in the drive

This routine assumes that when a disk is not in the drive, the write-protect is true. This is indeed the case for 3 inch and 8 inch disk drives, but is not the case for 5 1/4 inch disk drives

ENTRY CONDITIONS

None

EXIT CONDITIONS

If unit [illegible] present

Carry true

Otherwise

Carry false

Always

A BC DE HL [illegible] trashed

All other registers preserved

DD DRIVE STATUS
017Eh (382)

Issue a sense drive status command

ENTRY CONDITIONS

C = Unit/head
bits 0 . 1 = unit
bit 2 = head
bits 3 . 7 = 0

EXIT CONDITIONS

A = ST3 (Status register 3 of μPD765A)
Always
F BC DE HL IX corrupt
All other registers preserved

DD EQUIPMENT
0181h (385)

Ask what type of drive this is (ie single/double track single/double sided)

Track information can only be determined once a disk has been seen and had its type identified during logging in

Side information can only be detected after a double sided disk has been seen and had its type identified during logging in

ENTRY CONDITIONS

C = Unit (0 1)
IX = Address of XDPB

EXIT CONDITIONS

A = Side/track information
bits 0 . 1 = side information
0 = [illegible]
1 = single sided
2 = double sided
bits 2 . 3 = track information
0 = unknown
1 = single track
2 = double track
Always
F BC DE HL IX corrupt
All other registers preserved

DD ENCODE
0184h (388)

Set the copy protection ENCODE subroutine

Copy protected disks have some of their track and sector numbers encoded on disk. Before each disk access the ENCODE subroutine is called to encode the the physical track and sector numbers.

These encoded track and sector numbers must match those in the sector identifier.

Note that tracks 0 - 2 on either side of a disk should not be encoded.

ENTRY CONDITIONS

A = Enable/disable
  00h (0) = disable
  FFh (255) = enable
HL = (If enabled) address of ENCODE subroutine

EXIT CONDITIONS

HL = Address of previous ENCODE subroutine (0 if none)
Always
  AF BC DE IX corrupt
  All other registers preserved

NOTE

The definition of the subroutine ENCODE is as shown ahead. Note that if you are substituting your own ENCODE subroutine the entry conditions are the conditions passed to your subroutine and the exit conditions are the values that your subroutine must produce and the registers you are allowed to corrupt.

ENCODE

ENTRY CONDITIONS

C = Unit/side
  bits 0 - 1 = unit
  bit 2 = side
  bits 3 - 7 =
D = Physical track
E = Physical sector
IX = Address of DPB

EXIT CONDITIONS

D = Encoded physical track
E = Encoded physical sector
Always
  AF corrupt
  All other registers preserved

DD L XDPB
0187h (391)

Initialise an XDPB for a given format

This routine does not [illegible] under the free [illegible]

ENTRY CONDITIONS

IX = Addre[illegible] of destination XDPB
HL = Addre[illegible] disk specification

EXIT CONDITIONS

If OK
[illegible]
A = [illegible]
DE = [illegible]
HL = [illegible]
If [illegible]
[illegible]
A = [illegible]
DE HL corrupt
Always
BC IX corrupt
All other rem[illegible]

DD L DPB
018Ah (394)

Initialise a DPB f[illegible]

This routine does n[illegible]

ENTRY CONDITIONS

IX = Addre[illegible] destinat[illegible]
HL = Addre[illegible] of source disk sp[illegible]

EXIT CONDITIONS

If OK
[illegible]
A = [illegible]
[illegible]
[illegible]
[illegible]
[illegible]
A = [illegible]
[illegible]
[illegible]
[illegible]
[illegible]

DDL SEEK
018Dh (397)

Seeks to required track

Retry if fail

ENTRY CONDITIONS

C = [illegible] head

[illegible]

[illegible]

[illegible]

[illegible]

IX = Address of XDPB

EXIT CONDITIONS

If OK

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

DDL READ
0190h (400)

Low level µPD[illegible]

Read data

[illegible]

[illegible]

[illegible]

[illegible]

Writes [illegible]

Reads data

Reads results

Motor must be running

ENTRY CONDITIONS

HL = Address of parameter block

EXIT CONDITIONS

HL = Address of result buffer in [illegible]

Always

AF BC DE IX [illegible] rupt

All other registers preserved

DD L WRITE
0193h (403)

Low level μPD765A write command

Write data

Write deleted data

Format a track

Scan equal

Scan low or equal

Scan high or equal

Parameter block format

Byte 0	Page for C00[illegible]
Bytes [illegible] 2	Add[illegible]
Bytes [illegible] 4	Number of byte[illegible]
Byte 5	Number [illegible]
Bytes 6	Comm[illegible]nd byte[illegible]

Writes commands

Writes data

Reads results

Motor must be running

ENTRY CONDITIONS

HL = Address of parameter block

EXIT CONDITIONS

[illegible]

DD L ON MOTOR
0196h (406)

[illegible]

Wait for the mot[illegible]

ENTRY CONDITIONS

None

EXIT CONDITIONS

[illegible]

DD L T OFF MOTOR
0199h (409)

[illegible]

ENTRY CONDITIONS

[illegible]

EXIT CONDITIONS

[illegible]

DD L OFF MOTOR
019Ch (412)

Turn off the [illegible]

ENTRY CONDITIONS

[illegible]

EXIT CONDITIONS

[illegible]

Part 28
Spectrum character set

Subjects covered...

- Control codes
- Characters
- Z80 assembler mnemonics

This is the complete Spectrum [illegible] Z80 machine [illegible] and assembly language mn[illegible] starting with CBh or EDh [illegible] in the two [illegible] between 48K and +3 (128K) [illegible] codes the 4-K [illegible] +3 [illegible]

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible] not used	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	PRINT [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[EDIT]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	curs right	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	curs left	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	curs up	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[DELETE]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[ENTER]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	number	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	INK [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	PAPER [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	FLASH [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	BRIGHT [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	INVERSE [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	OVER [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	AT [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	TAB [illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
[illegible]	[illegible]	[illegible]	ld a (de)	rr d	
[illegible]	[illegible]	[illegible]	dec de	rr e	
[illegible]	not used	[illegible]	inc e	rr h	
[illegible]	[illegible]	[illegible]	dec e	rr l	
[illegible]	[illegible]	[illegible]	ld e,N	rr (hl)	
[illegible]	[illegible]	[illegible]	rra	rr a	
[illegible]	[illegible]	[illegible]	jr nz DIS	sla b	
[illegible]	!	[illegible]	ld hl,NN	sla c	
[illegible]	"	[illegible]	ld (NN) hl	sla d	
[illegible]	#	[illegible]	inc hl	sla e	
[illegible]	$	[illegible]	inc h	sla h	
[illegible]	%	[illegible]	dec h	sla l	
[illegible]	&	[illegible]	ld h,N	sla (hl)	
[illegible]	'	[illegible]	daa	sla a	
[illegible]	(	[illegible]	jr z DIS	sra b	
[illegible]	)	[illegible]	add hl,hl	sra c	
[illegible]	*	[illegible]	ld hl (NN)	sra d	
[illegible]	+	[illegible]	dec hl	sra e	
[illegible]	,	[illegible]	inc l	sra h	
[illegible]	-	[illegible]	dec l	sra l	
[illegible]	.	[illegible]	ld l,N	sra (hl)	
[illegible]	/	[illegible]	cpl	sra a	
[illegible]	0	[illegible]	jr nc DIS		
[illegible]	1	[illegible]	ld sp,NN		
[illegible]	2	[illegible]	ld (NN) a		
[illegible]	3	[illegible]	inc sp		
[illegible]	4	[illegible]	inc (hl)		
[illegible]	5	[illegible]	dec (hl)		
[illegible]	6	[illegible]	ld (hl),N		
[illegible]	7	[illegible]	scf		
[illegible]	8	[illegible]	jr c DIS	[illegible]	
[illegible]	9	[illegible]	add hl,sp	[illegible]	
[illegible]	:	[illegible]	ld a (NN)	[illegible]	
[illegible]	;	[illegible]	[illegible]	[illegible]	
[illegible]	<	[illegible]	[illegible]	[illegible]	
[illegible]	=	[illegible]	[illegible]	[illegible]	
[illegible]	>	[illegible]	ld e,N	[illegible]	
[illegible]	?	[illegible]	[illegible]	[illegible]	
[illegible]	@	[illegible]	ld b,b	[illegible]	in b,(c)
[illegible]	A	[illegible]	ld b,c	[illegible]	out (c),b
[illegible]	B	[illegible]	ld b,d	[illegible]	sbc hl,bc
[illegible]	C	[illegible]	ld b,e	[illegible]	ld (NN) bc
[illegible]	D	[illegible]	ld b,h	[illegible]	neg
[illegible]	E	[illegible]	ld b,l	[illegible]	retn
[illegible]	F	[illegible]	ld b,(hl)	[illegible]	im [illegible]
[illegible]	G	[illegible]	ld b,a	[illegible]	[illegible]
[illegible]	H	[illegible]	ld c,b	[illegible]	[illegible]

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
73	I	49	ld c,c	bit 1,c	out (c),c
74	J	4A	ld c,d	bit 1,d	adc hl,bc
75	K	4B	ld c,e	bit 1,e	ld bc,(NN)
76	L	4C	ld c,h	bit 1,h	
77	M	4D	ld c,l	bit 1,l	reti
78	N	4E	ld c,(hl)	bit 1,(hl)	
79	O	4F	ld c,a	bit 1,a	ld r,a
80	P	50	ld d,b	bit 2,b	in d,(c)
81	Q	51	ld d,c	bit 2,c	out (c),d
82	[illegible]	52	ld d,d	bit 2,d	sbc hl,de
83	S	53	ld d,e	bit 2,e	ld (NN),de
84	T	54	ld d,h	bit 2,h	
85	U	55	ld d,l	bit 2,l	
86	V	56	ld d,(hl)	bit 2,(hl)	im 1
87	W	57	ld d,a	bit 2,a	ld a,i
88	X	58	ld e,b	bit 3,b	in e,(c)
89	Y	59	ld e,c	bit 3,c	out (c),e
90	Z	5A	ld e,d	bit 3,d	adc hl,de
91	[	5B	ld e,e	bit 3,e	ld de,(NN)
92	\	5C	ld e,h	bit 3,h	
93	]	5D	ld e,l	bit 3,l	
94	↑	5E	ld e,(hl)	bit 3,(hl)	im 2
95	_	5F	ld e,a	bit 3,a	ld a,r
96	£	60	ld h,b	bit 4,b	in h,(c)
97	a	61	ld h,c	bit 4,c	out (c),h
98	b	62	ld h,d	bit 4,d	sbc hl,hl
99	c	63	ld h,e	bit 4,e	ld (NN),hl
100	d	64	ld h,h	bit 4,h	
101	e	65	ld h,l	bit 4,l	
102	f	66	ld h,(hl)	bit 4,(hl)	
103	g	67	ld h,a	bit 4,a	rrd
104	h	68	ld l,b	bit 5,b	in l,(c)
105	i	69	ld l,c	bit 5,c	out (c),l
106	j	6A	ld l,d	bit 5,d	adc hl,hl
107	k	6B	ld l,e	bit 5,e	ld hl,(NN)
108	l	6C	ld l,h	bit 5,h	
109	m	6D	ld l,l	bit 5,l	
110	n	6E	ld l,(hl)	bit 5,(hl)	
111	o	6F	ld l,a	bit 5,a	rld
112	p	70	ld (hl),b	bit 6,b	in f,(c)
113	q	71	ld (hl),c	bit 6,c	
114	r	72	ld (hl),d	bit 6,d	sbc hl,sp
115	s	73	ld (hl),e	bit 6,e	ld (NN),sp
116	t	74	ld (hl),h	bit 6,h	
117	u	75	ld (hl),l	bit 6,l	
118	v	76	halt	bit 6,(hl)	
119	w	77	ld (hl),a	bit 6,a	

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
[illegible]	x	[illegible]	ld a,b	bit 7,b	in a,(c)
[illegible]	y	[illegible]	ld a,c	bit 7,c	out (c),a
[illegible]	z	[illegible]	ld a,d	bit 7,d	adc hl,sp
[illegible]	{	[illegible]	ld a,e	bit 7,e	ld sp,(NN)
[illegible]	|	[illegible]	ld a,h	bit 7,h	
[illegible]	}	[illegible]	ld a,l	bit 7,l	
[illegible]	[illegible]	[illegible]	ld a,(hl)	bit 7,(hl)	
[illegible]	[illegible]	[illegible]	ld a,a	bit 7,a	
[illegible]	[illegible]	[illegible]	add a,b	res [illegible]	
[illegible]	[illegible]	[illegible]	add a,c	[illegible]	
[illegible]	[illegible]	[illegible]	add a,d	[illegible]	
[illegible]	[illegible]	[illegible]	add a,e	[illegible]	
[illegible]	[illegible]	[illegible]	add a,h	[illegible]	
[illegible]	[illegible]	[illegible]	add a,l	res 0,l	
[illegible]	[illegible]	[illegible]	add a,(hl)	res 0,(hl)	
[illegible]	[illegible]	[illegible]	add a,a	res 0,a	
[illegible]	[illegible]	[illegible]	adc a,b	res [illegible]	
[illegible]	[illegible]	[illegible]	adc a,c	res 1,c	
[illegible]	[illegible]	[illegible]	adc a,d	res 1,d	
[illegible]	[illegible]	[illegible]	adc a,e	res 1,e	
[illegible]	[illegible]	[illegible]	adc a,h	[illegible]	
[illegible]	[illegible]	[illegible]	adc a	[illegible]	
[illegible]	[illegible]	[illegible]	adc a,(hl)	[illegible]	
[illegible]	[illegible]	[illegible]	adc a,a	[illegible]	
[illegible]	(a)	[illegible]	sub b	[illegible]	
[illegible]	(b)	[illegible]	sub c	[illegible]	
[illegible]	(c)	[illegible]	sub d	[illegible]	
[illegible]	(d)	[illegible]	sub e	[illegible]	
[illegible]	(e)	[illegible]	sub h	[illegible]	
[illegible]	(f)	[illegible]	sub	res 2,l	
[illegible]	(g)	[illegible]	sub (hl)	res 2,(hl)	
[illegible]	(h)	[illegible]	sub a	res 2,a	
[illegible]	(i)	[illegible]	sbc a,b	res 3,b	
[illegible]	(j)	[illegible]	sbc	res 3,c	
[illegible]	(k)	[illegible]	sbc a,d	res 3,d	
[illegible]	(l)	[illegible]	sbc a,e	[illegible]	
[illegible]	(m)	[illegible]	sbc a,h	[illegible]	
[illegible]	(n)	[illegible]	sbc a	[illegible]	
[illegible]	(o)	[illegible]	sbc a,(hl)	[illegible]	
[illegible]	(p)	[illegible]	sbc a,a	[illegible]	
[illegible]	(q)	[illegible]	[illegible]	[illegible]	ldi
[illegible]	(r)	[illegible]	[illegible]	[illegible]	cpi
[illegible]	(s)	[illegible]	[illegible]	[illegible]	ini
[illegible]	SPECTRUM (t)	[illegible]	and e	[illegible]	outi
[illegible]	PLAY (u)	[illegible]	and h	[illegible]	
[illegible]	RND	[illegible]	and l	[illegible]	
[illegible]	INKEY$	[illegible]	and (hl)	[illegible]	

graphics set

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
167	PI	A7	and a	[illegible]	
168	FN	A8	xor b	[illegible]	[illegible]
169	POINT	A9	xor c	[illegible]	[illegible]
170	SCREEN$	AA	xor d	[illegible]	[illegible]
171	ATTR	AB	xor e	[illegible]	[illegible]
172	AT	AC	xor h	[illegible]	
173	TAB	AD	xor l	[illegible]	
174	VAL$	AE	xor (hl)	[illegible]	
175	CODE	AF	xor a	[illegible]	
176	VAL	B0	or b	[illegible]	[illegible]
177	LEN	B1	[illegible]	[illegible]	[illegible]
178	SIN	B2	[illegible]	[illegible]	[illegible]
179	COS	[illegible]	[illegible]	[illegible]	[illegible]
180	TAN	B4	[illegible]	[illegible]	
181	ASN	B5	[illegible]	[illegible]	
182	ACS	B6	[illegible]	[illegible]	
183	ATN	[illegible]	[illegible]	[illegible]	
184	LN	B8	[illegible]	[illegible]	[illegible]
185	EXP	B9	[illegible]	[illegible]	[illegible]
186	INT	BA	[illegible]	[illegible]	[illegible]
187	SQR	BB	[illegible]	[illegible]	[illegible]
188	SGN	[illegible]	[illegible]	[illegible]	
189	ABS	BD	[illegible]	[illegible]	
190	PEEK	BE	[illegible]	[illegible]	
191	IN	BF	[illegible]	[illegible]	
192	USR	[illegible]	[illegible]	[illegible]	
193	STR$	C1	[illegible]	[illegible]	
194	CHR$	C2	[illegible]	[illegible]	
195	NOT	C3	[illegible]	[illegible]	
196	BIN	C4	[illegible]	[illegible]	
197	OR	C5	[illegible]	[illegible]	
198	AND	C6	[illegible]	[illegible]	
199	<=	C7	[illegible]	[illegible]	
200	>=	C8	[illegible]	[illegible]	
201	<>	C9	[illegible]	[illegible]	
202	LINE	CA	[illegible]	[illegible]	
203	THEN	CB		[illegible]	
204	TO	CC	[illegible] NN	[illegible]	
205	STEP	CD	call NN	[illegible]	
206	DEF FN	CE	[illegible]	[illegible]	
207	CAT	CF	[illegible]	[illegible]	
208	FORMAT	D0	ret nc	[illegible]	
209	MOVE	D1	pop de	[illegible]	
210	ERASE	D2	jp nc NN	[illegible]	
211	OPEN #	D3	[illegible]	[illegible]	
212	CLOSE #	D4	[illegible]	[illegible]	
213	MERGE	[illegible]	[illegible]	[illegible]	

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
214	VERIFY	[illegible]	[illegible]	set 2,(hl)	
[illegible]	BEEP	[illegible]	[illegible]	set 2,a	
216	CIRCLE	[illegible]	[illegible]	set 3,b	
[illegible]	INK	[illegible]	[illegible]	set 3,c	
[illegible]	PAPER	[illegible]	[illegible]	set 3,d	
219	FLASH	[illegible]	[illegible]	set 3,e	
220	BRIGHT	[illegible]	[illegible]	set 3,h	
221	INVERSE	[illegible]	[illegible]	set 3,l	
			instructions		
			using ix		
[illegible]	OVER	[illegible]	sbc a,N	[illegible]	
[illegible]	OUT	[illegible]	rst 24	[illegible]	
[illegible]	LPRINT	[illegible]	ret nc	[illegible]	
[illegible]	LLIST	[illegible]	pop hl	[illegible]	
[illegible]	STOP	[illegible]	jp po,NN	[illegible]	
[illegible]	READ		ex (sp),hl	set 4,e	
228	DATA	[illegible]	[illegible]	set 4,h	
229	RESTORE		[illegible]	set 4	
230	NEW	[illegible]	[illegible]	set 4,(hl)	
231	BORDER	[illegible]	[illegible]	[illegible]	
232	CONTINUE	[illegible]	[illegible]	[illegible]	
233	DIM		[illegible]	[illegible]	
234	REM	[illegible]	[illegible]	[illegible]	
235	FOR	[illegible]	[illegible]	[illegible]	
[illegible]	GO TO	[illegible]	[illegible]		
[illegible]	GO SUB	[illegible]		[illegible]	
238	INPUT	[illegible]	call N	[illegible]	
239	LOAD	[illegible]	rst 40		
240	LIST	[illegible]	ret p	[illegible]	
241	LET		pop af	[illegible]	
242	PAUSE		jp p,NN	[illegible]	
243	NEXT			[illegible]	
244	POKE	[illegible]	[illegible]	[illegible]	
245	PRINT	[illegible]	[illegible]	[illegible]	
246	PLOT	[illegible]	[illegible]	set 6,(hl)	
247	RUN	[illegible]	[illegible]	set 6,a	
[illegible]	SAVE	[illegible]	[illegible]	[illegible]	
249	RANDOMIZE	[illegible]	[illegible]	[illegible]	
250	IF	[illegible]	[illegible]	[illegible]	
[illegible]	CLS	[illegible]		[illegible]	
[illegible]	DRAW	[illegible]	[illegible]	[illegible]	
[illegible]	CLEAR	[illegible]	[illegible]	[illegible]	
			[illegible]		
			[illegible]		
[illegible]	RETURN	[illegible]	[illegible]	[illegible]	
[illegible]	COPY	[illegible]	[illegible]	[illegible]	

Part 29
Reports

Subjects covered...

Reports and messages
CONTINUE

Reports appear at the bottom of the screen whenever the +3 has st[illegible] They explain why it has stopped – be it for so[illegible] has occurred.

Most reports have [illegible] number or letter [illegible] a brief message explaining what happ[illegible] d the [illegible] in the line w[illegible] 0 [illegible] 1 [illegible] at the begin[illegible] statement 2 [illegible] THEN [illegible]

Reports pertaining to disk operation (or +3DOS) do not start [illegible] or letter – they are show[illegible] ahead in alphabetical order.

The behaviour of the CONTINUE [illegible] mand depends v[illegible] ich on the report. Normally CONTINUE goes to the line [illegible] statement specified i[illegible] port but there ar[illegible] tions with reports 0, 9 and D.

Here is a table showin[illegible] all the reports. The [illegible] ances the report can occur and th[illegible] to part 31 [illegible] from the table that the error A Invalid argument [illegible] SQR LN ACS ASN [illegible] USR 0 you then look up these keywor[illegible] and [illegible] ch argument *are* invalid.

Disk errors marked by RIC [illegible] the left-hand volume [illegible] - Retry, Ignore or Cancel? [illegible] the report [illegible] in the second [illegible] will [illegible]

CODE	REPORT/EXPLANATION	SITUATION
0	OK [illegible] and statement [illegible] CONTINUE	Any
1	NEXT without FOR The [illegible] FOR [illegible] *is* [illegible]	NEXT

CODE	REPORT/EXPLANATION	SITUATION
2	Variable not found LET READ INPUT FOR DIM	
3	Subscript wrong B	ariable
4	Out of memory DELETE	LET INPUT FOR DIM GO SUB LOAD MERGE
5	Out of screen A INPUT PRINT AT 22,x	INPUT PRINT AT
6	Number too big Calculations have yielde proxima	Any arithmeti
7	RETURN without GO SUB RETURN GO SUB	RETURN
9	STOP statement Also this CONTINUE STOP	STOP
A	Invalid argument	SQR LN ASN ACS USR
B	Integer out of range	RUN RANDOMIZE POKE DIM GO TO GO SUB LIST LLIST PAUSE PLOT CHR$ PEEK USR numeric

CODE	REPORT/EXPLANATION	SITUATION
C	Nonsense in BASIC [illegible]	VAL VAL$
D	BREAK - CONT repeats **BREAK** [illegible] CONTINUE [illegible] [illegible] L	LOAD SAVE VERIFY MERGE After [illegible] that the [illegible] scroll? [illegible] [illegible] **N BREAK** or the [illegible]
E	Out of DATA You'll [illegible] to READ [illegible] DATA [illegible]	READ
F	Invalid filename SAVE will [illegible] empty [illegible] character [illegible]	SAVE
G	No room for line [illegible]	[illegible]
H	STOP in INPUT [illegible] INPUT data [illegible] STOP [illegible] [illegible] 9 [illegible] CONTINUE [illegible] [illegible] INPUT state [illegible]	INPUT
I	FOR without NEXT [illegible] FOR [illegible] FOR n=1 TO 0 [illegible] NEXT [illegible]	FOR
J	Invalid I/O device [illegible] INPUT #2,a$ [illegible]	[illegible] OPEN # CLOSE # INPUT # PRINT #

CODE	REPORT/EXPLANATION	SITUATION
K	Invalid colour The number [illegible]	INK PAPER BORDER FLASH BRIGHT INVERSE OVER also after one of the corresponding control characters
L	BREAK into program BREAK [illegible] [illegible] BREAK [illegible] CONTINUE [illegible]	[illegible]
M	RAMTOP no good [illegible]	CLEAR [illegible] RUN
N	Statement lost [illegible]	RETURN NEXT CONTINUE
O	Invalid Stream [illegible]	INPUT # OPEN # PRINT #
P	FN without DEF [illegible] DEF [illegible]	FN
Q	Parameter error [illegible]	FN
R	Tape loading error [illegible] read [illegible]	VERIFY LOAD MERGE
d	Too many brackets [illegible]	PLAY
j	Invalid baud rate [illegible]	FORMAT LINE
k	Invalid note name PLAY [illegible]	PLAY

CODE	REPORT/EXPLANATION	SITUATION
l	Number too big A parameter for a [illegible] is too big	PLAY
m	Note out of range A series of sh[illegible]	PLAY
n	Out of range A [illegible]	PLAY
o	Too many tied notes An [illegible]	PLAY
	Bad filename The [illegible]	CAT COPY ERASE FORMAT LOAD MERGE MOVE SAVE
	Bad parameters [illegible] range. It is unlikely th[illegible]	Unlikely
RIC	CRC data error The [illegible]	CAT COPY ERASE LOAD MERGE MOVE SAVE
	Code length error Trying [illegible] CODE file from [illegible] the value [illegible] LOAD [illegible]	LOAD CODE
	Destination cannot be wild Try [illegible] and [illegible] COPY [illegible] *only* [illegible]	COPY TO
	Destination must be drive Does [illegible] COPY [illegible] *only* [illegible]	COPY TO

CODE	REPORT/EXPLANATION	SITUATION
	Directoryfull	COPY SAVE
	Disk full CAT	COPY SAVE
RIC	Disk has been changed +3	CAT COPY ERASE LOAD MERGE MOVE SAVE
	Disk is not bootable	LOAD "*"
RIC	Disk is write protected	COPY ERASE FORMAT MOVE SAVE
	Drive B: is not present the FORMAT	FORMAT
	Drive in use	
	Drive not found ERASE "c:fred"	CAT COPY ERASE LOAD MERGE MOVE SAVE

CODE	REPORT/EXPLANATION	SITUATION
RIC	**Drive not ready** A disk [illegible] the drive [illegible] ns because there is no disk in the drive [illegible] usually be possible to simply put a disk in the drive and type R	CAT COPY ERASE FORMAT LOAD MERGE MOVE SAVE
	End of file found An attempt has been made to read a byte past the end-of-file position. It is unlikely that this will be seen	Unlikely
	File already exists The destination filename [illegible] MOVE [illegible]	MOVE TO
	File already in use [illegible] +3 [illegible] that this error [illegible]	Unlikely COPY LOAD MERGE SAVE
	File is read only Trying to [illegible] save using the name [illegible] that has its protection turned on [illegible] using the [illegible] MOVE filename TO "+P" [illegible] MOVE [illegible] TO "-P" [illegible] prot[illegible]	COPY ERASE MOVE SAVE
	File not found The filename given for one of the disk [illegible] file that for [illegible]	COPY ERASE LOAD MERGE MOVE
	File not open [illegible]	Unlikely
	File too big [illegible]	[illegible]
	Invalid attribute [illegible] MOVE [illegible] P S :A [illegible]	MOVE TO
	Invalid drive A drive is [illegible] A: or B: [illegible] FORMAT command [illegible]	FORMAT

CODE	REPORT EXPLANATION	SITUATION
RIC	Missing address mark	CAT COPY ERASE LOAD MERGE MOVE SAVE
	Missing extent extent	COPY LOAD MERGE
RIC	No data	CAT COPY ERASE LOAD MERGE MOVE SAVE
	No rename between drives MOVE	MOVE TO
RIC	Seek fail	CAT COPY ERASE FORMAT LOAD MERGE MOVE SAVE
	Uncached	
RIC	Unknown disk error	CAT COPY ERASE FORMAT LOAD MERGE MOVE SAVE

CODE	REPORT/EXPLANATION	SITUATION
RIC	Unrecognised disk format [illegible]	CAT COPY ERASE LOAD MERGE MOVE SAVE
RIC	Unsuitable media The [illegible] Thi [illegible] +3	CAT COPY ERASE FORMAT LOAD MERGE MOVE SAVE

Part 30
Reference information

Subjects covered...

Hardware

The ***+3*** is designed around the Z80A microprocessor, which runs at a speed of 3.5469 MHz (about three and half million cycles per second).

The ***+3*** s memory is divided into 64K ROM and 128K RAM, arranged in 16K pages. The four ROM pages (0-3) can be mapped into the bottom 16K (0000h-3FFFh) of the memory map. The eight RAM pages (0-7) are usually mapped into the top 16K (C000h-FFFFh) of the memory map. RAM page 5 is also mapped into the range 4000h-7FFFh, and RAM page 2 is mapped into the range 8000h-BFFFh. There are also several RAM page combinations that occupy the full 64K address range. These were given in part 24 of this chapter, under the heading 'Memory management'.

Physically speaking, the ROMs are two 32K devices (similar to the 27256), which are both treated by the system as two 16K chips. The RAM is composed of four 64K x 4 bit chips (41464), some of which (RAM banks 4-7) are time-shared between the circuitry that produces the screen display, and the Z80A. The others (RAM banks 0-3) are for the exclusive use of the Z80A, as is the ROM.

For the contended RAM (which shares time between the video circuitry and the processor), during 128 out of every 228 CPU T states (1 TV line), and during 192 out of every 311 TV lines (1 frame), the CPU is allowed only 1 access to contended RAM in every 8 T states. The CPU is controlled by introducing wait states.

Executing NOP instructions in contended RAM will have an effective average clock frequency of 2.66MHz (a reduction of about 25%).

The Uncommitted Logic Array (**ULA**) handles most of the I/O such as keyboard, tape interface, part of the printer interface and screen handling. It converts bytes in memory into patterns and colours on the screen, and allows the Z80A to scan the keyboard and read and write data to tape.

The three-channel sound is produced by the AY-3-8912 - a very popular sound chip, and this device also controls the **RS232/MIDI** and **AUX** ports.

The two serial ports can be driven only by software. The ***+3*** has no software support for the **AUX** port - this is left to the user's discretion. The **RS232/MIDI** port is fully supported from ***+3*** BASIC.

The way in which the AY-3-8912 works is quite complex, and the would-be experimenter is advised to get the manufacturer's data sheet. The following information should be enough to get things underway, however.

The sound chip contains sixteen registers which are selected by writing first to the address write port FFFDh (65533) with the register number, and then reading the register value (same address), or writing to the data register write address BFFDh (49149). Once a register has been selected, any number of data read/writes can be made; the address write port need only be re-written if a different register needs to be accessed.

The basic clock frequency of the circuit is [illegible] MHz (to 0.01 %)

The registers do the following

R0 - Fine tone control channel A
R1 - Coarse tone control channel A
R2 - Fine tone control channel B
R3 - Coarse tone control channel B
R4 - Fine tone control channel C
R5 - Coarse tone control channel C

The tone of a channel is a 12 bit value taken from the sum of D3-D0 of the coarse register and D7-D0 of the the fine register. The basic unit of tone is the clock frequency divided by 16 (ie. 110.83 KHz) and with a 12 bit counter range, frequencies from 27Hz to 110 KHz can be generated.

R6 - Noise generator control D4-D0

The period of the noise source is taken by counting down the lower 5 bits of the noise register every sound clock period divided by 16

R7 - Mixer and I/C control

D7 - Not used
D6 - 1 means input port
- 0 means output port
D5 - Channel C noise
D4 - Channel B noise
D3 - Channel A noise
D2 - Channel C tone
D1 - Channel B tone
D0 - Channel A tone

This register controls both the mixing of noise and tone values for each channel and the direction of the 8 bit I/O port. A zero in a mix bit indicates that the function is *enabled*

R8 - Amplitude control channel A
R9 - Amplitude control channel B
RA - Amplitude control channel C

D4 - 1 means use envelope generator
- 0 means use value of D3-D0 for amplitude
D3-D0 - Amplitude

These three registers control the amplitude of each channel and whether or not it is modulated by the envelope registers

RB - Envelope coarse period control
RC - Envelope fine period control

The eight bit values in RB+RC are summed to produce a 16 bit number which is counted down in units of 256 multiplied by the sound clock. Envelope frequencies can be between 0.1Hz and 6KHz.

RD - Envelope control

D3 - Continue
D2 - Attack
D1 - Alternate
D0 - Hold

The diagram of envelope shapes (in part 19 of this chapter) gives a graphic illustration of the possible settings for this register.

The disk drive is controlled by the μPD765A floppy disk controller chip. As described in part 23 of this chapter, the data register for this device is at address 3FFDh (16381) and the status register is at 2FFDh (12285). This is a very complex device and it would be unwise to attempt to use it without full details of its operation (see the manufacturer's data sheet).

The Centronics parallel printer port is basically just an 8 bit data latch (74273) whose address is 0FFDh (4093). The STROBE signal for the the printer is produced by the ULA and is accessed using bit 4 of address 1FFDh (8189). The state of the BUSY line from the printer is read from bit 7 of address 0FFDh (4093).

Part 31
The BASIC

Subjects covered...

- Number handling
- Variables
- Strings
- Functions
- Brief summary of keywords
- Mathematical operations

Numbers are stored to an accuracy of 9 or 10 digits. The largest number you can get is about $10^{[illegible]}$ and the smallest (positive) number [illegible] about 4 x 10[illegible]

Unless a number represents an exact power of 3 there is a possibility that mathematical inaccuracies may become apparent after repeated addition, subtraction, etc. This is true of all computers that do not use BCD arithmetic. Use of integers is suggested if absolute mathematical accuracy [illegible] required.

A number is stored in the [illegible] +3 in floating point binary with one exponent byte e ([illegible] <=e<=255) and four mantissa bytes m ([illegible] <=m<[illegible]). This represents the number m x 2[illegible]

Since [illegible] <=m<1, the most significant bit of the mantissa m is always [illegible]. Therefore, in actual fact we can replace it with a bit to show the sign: 0 for positive numbers, [illegible] for negative.

Small integers have a special representation in which the first byte is 30h [illegible] the second is a sign byte (00h or FFh) and the third and fourth are the integer itself (in twos complement form) with the least significant byte first.

Numeric variables have names of arbitrary length, starting with a letter and continuing with letters [illegible]

[illegible] FOR NEXT [illegible] letter long.

[illegible] be the [illegible] name of a simple variable. They may have many dimensions of arbitrary size. Subscript va[illegible]

Strings are completely flexible in length. The name of a string consists of a single letter [illegible] ed by $.

String arrays can have many dimensions of [illegible] $ [illegible] may *not* be the same as the name of a sim[illegible] array have the same fixed length, which is specified as an extra [illegible] DIM [illegible] Subscript values start at 1.

Slicing substrings of strings may be specified using [illegible]

(i) empty

.

(ii) a [illegible]

or

(m) [illegible] expression TO [illegible]

and is used in expressing a substring by either

(a) string expression (slice)

or

(b) string array variable [illegible]

which is the same as

string array [illegible]

If m is suppose the string [illegible] s$ [illegible] empty the result is s$ (considered as a substring [illegible]

If the slicer is a [illegible] the mth character of s$ is substring of length 1.

If [illegible] value is [illegible] s$ [illegible] <= [illegible] <= [illegible] s$, then [illegible] s$ [illegible]

If 0 <= n <= [illegible]

Slicing is [illegible] functions or [illegible] (otherwise)

Substring [illegible] (see LET). If a string [illegible] numeral [illegible] it must be doubled.

Functions

[illegible] constant or a variable [illegible] optionally [illegible]

FUNCTION	TYPE OF ARGUMENT	RESULT
ABS	[illegible]	[illegible]
ACS	[illegible]	[illegible] Arcs [illegible]

FUNCTION	TYPE OF ARGUMENT	RESULT
AND	[illegible]	a AND b — a [illegible] b<> [illegible] b=
	[illegible]	a$ AND b — a$ [illegible] b<> [illegible] b=
		AND [illegible]
ASN	[illegible]	[illegible] A [illegible]
ATN	[illegible]	[illegible]
ATTR	[illegible]	[illegible] B [illegible] <= [illegible] <= [illegible] <= [illegible] <= [illegible]
BIN	[illegible]	[illegible] BIN [illegible]
CHR$	[illegible]	[illegible]
CODE	[illegible]	[illegible]
COS	[illegible]	[illegible]
EXP	[illegible]	

FUNCTION	TYPE OF ARGUMENT	RESULT
FN		FN [illegible] DEF [illegible]
IN	[illegible]	[illegible] <=,<= [illegible] instruction ■ a,(C)
INKEY$	[illegible]	read the keyboard. The result [illegible] letter representing the [illegible] and there is exactly [illegible]
INT	[illegible]	[illegible]
LEN	[illegible]	[illegible]
LN	[illegible]	[illegible] A [illegible] <=
NOT	[illegible]	[illegible] <> [illegible] = NOT [illegible]
OR	[illegible]	a OR b → [b<> / a b=] OR [illegible]
PEEK	[illegible]	[illegible] B,A [illegible]
PI	[illegible]	[illegible]
POINT	[illegible]	[illegible] B [illegible] <=,<= [illegible] <=,<= [illegible]

FUNCTION	TYPE OF ARGUMENT	RESULT
RND	[illegible]	[illegible] <=,<
SCREEN$	[illegible]	[illegible] B [illegible] <=,<= [illegible] <=,<=
SGN	[illegible]	[illegible] - [illegible] + [illegible]
SIN	[illegible]	[illegible]
SQR	[illegible]	[illegible] A <
STR$	[illegible]	[illegible]
TAN	[illegible]	[illegible]
USR	[illegible]	[illegible]

FUNCTION	TYPE OF ARGUMENT	RESULT
USR	[illegible]	[illegible]
VAL	[illegible]	[illegible]
VAL$	[illegible]	[illegible] VAL
-	[illegible]	[illegible]

[illegible]

+ [illegible]
- [illegible]
* [illegible]
/ [illegible]
^ [illegible] B [illegible]
= [illegible]
> [illegible]
< [illegible]
<= [illegible]
>= [illegible]
<> [illegible]

Functions and [illegible] priorities

OPERATION	PRIORITY
[illegible]	[illegible]
A[illegible] NOT [illegible]	[illegible]
[illegible]	[illegible]
- [illegible]	[illegible]
* / [illegible]	[illegible]
+ - [illegible]	[illegible]
= > < <= >= <> [illegible]	[illegible]
NOT	[illegible]
AND	
OR	

Statements

[illegible]

[illegible]	[illegible]
[illegible]	[illegible]
X Y Z	[illegible]
m n	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible] A: B: M: T:
[illegible]	[illegible]
[illegible]	[illegible] contain the [illegible] * ?
[illegible]	[illegible]
[illegible]	rep[illegible] pu[illegible] [illegible] of [illegible] colons A [illegible] PAPER INK FLASH BRIGHT INVERSE [illegible] OVER statement

Note that optic[illegible]

Arbitrary express[illegible]

A[illegible] INPUT DEF FN [illegible] DATA [illegible] IF [illegible] REM

BEEP x,y	[illegible]
BORDER [illegible]	[illegible]
BRIGHT n	[illegible] sequently printed [illegible]
CAT #[illegible]	[illegible] CAT [illegible] phanumerically [illegible] in the form CAT # [illegible] CAT [illegible] T: [illegible] to tape [illegible] with [illegible]
CAT (#[illegible], id = EXP	[illegible] CAT [illegible] and [illegible] MOVE [illegible] TO [illegible]
CIRCLE x,y,r	[illegible] radius
CLEAR	[illegible] they previously [illegible] RESTORE [illegible] CLS [illegible] PLOT [illegible] GO SUB [illegible]
CLEAR	[illegible] CLEAR [illegible] GO SUB [illegible]
CLOSE #n	[illegible] OPEN #[illegible]
CLS	[illegible]

CONTINUE	[illegible] 0 [illegible] 9 [illegible] CONTINUE [illegible] 0:1 [illegible] 0 [illegible] 0:2 [illegible] N [illegible] 0:3 [illegible]
COPY	[illegible] D BREAK [illegible]
COPY EXP INVERSE	[illegible] BRIGHT [illegible] INVERSE [illegible] D BREAK [illegible]
COPY u TO [illegible] COPY [illegible] TO [illegible] COPY [illegible] TO [illegible]	[illegible] only [illegible] +3 [illegible] File already exists [illegible] Missing address mark [illegible]

COPY u TO SCREEN$	[illegible]
COPY u TO LPRINT	[illegible] of the [illegible] to the printer [illegible] character trans[illegible] FORMAT LPRINT "R" [illegible] RS232 [illegible] this form [illegible] COPY [illegible]
COPY . TO SPECTRUM FORMAT	[illegible] .HED [illegible]
DATA , , ,	Part of the DATA [illegible] be in a [illegible] otherwise [illegible]
DEF FN (,) =	[illegible] Mu[illegible] program [illegible] $ [illegible] DEF FN ()= [illegible]
DIM (,)	[illegible] a array for [illegible]
DIM $(,)	[illegible] $ [illegible] $ [illegible] ned by DIM [illegible] 4 there is no room [illegible] the [illegible]
DRAW x,y	DRAW x,y,0
DRAW x,y,z	[illegible] while turning [illegible] B [illegible]
ERASE ERASE .	[illegible] Y [illegible] ERASE [illegible]

FLASH	[illegible]
FOR [illegible] TO [illegible]	FOR [illegible] TO [illegible] STEP 1
FOR [illegible] TO [illegible] STEP [illegible]	[illegible] FOR [illegible] >= [illegible] < [illegible] NEXT [illegible] 1 [illegible] NEXT [illegible]
FORMAT	[illegible] A: [illegible] B: [illegible] +3 [illegible]
FORMAT LINE [illegible]	[illegible]
FORMAT LPRINT [illegible] ;	[illegible] "C" [illegible] PRINTER [illegible] "R" [illegible] RS232 [illegible] "E" [illegible] CHRS 32 [illegible] CHRS 127 [illegible] "U" [illegible] "C" [illegible] "R" [illegible] "E" [illegible] "U"
GO SUB	[illegible] GO SUB [illegible] GO TO [illegible] RETURN
GO TO [illegible]	[illegible]
IF [illegible] THEN	[illegible] IF [illegible] THEN [illegible]

INK n	[illegible] K [illegible] <=:<=
INPUT [#n,]	[illegible] INPUT [illegible] separated [illegible] PRINT [illegible] INPUT [illegible] PRINT [illegible] LINE [illegible] PRINT [illegible] PRINT [illegible] the [illegible] and [illegible] type [illegible] contain tw [illegible] erased [illegible] the [illegible] STOP SYMB SHIFT [illegible] A [illegible] H [illegible] rary [illegible] STOP [illegible]
INVERSE [illegible]	[illegible] = [illegible] = [illegible] K [illegible] = [illegible] INV VIDEO [illegible] INVERSE 1 [illegible] the TRUE VIDEO [illegible] INVERSE [illegible]
LET [illegible] = [illegible]	[illegible] LET [illegible] not be [illegible] LET READ INPUT [illegible]
LIST [#n]	LIST #n, 0

LIST [#m, n	[illegible] starting [illegible] in the [illegible] #m [illegible] ent to the [illegible]
LLIST	LLIST 0
LLIST	[illegible] LIST [illegible] **PRINTER** [illegible] – attached to the **RS232** [illegible] FORMAT LPRINT "R" [illegible] acht [illegible] FORMAT LPRINT "E" [illegible] restor this state if it has [illegible] by FORMAT LPRINT "U"
LOAD	[illegible] detail in device [illegible] COPY ERASE MOVE [illegible] T: the [illegible] LOAD [illegible]
LOAD	[illegible] LOAD "*" [illegible]
LOAD DATA ()	[illegible] () [illegible]
LOAD DATA $()	[illegible] $() [illegible]
LOAD CODE n,[illegible]	[illegible]
LOAD CODE	[illegible] COPY TO SPECTRUM FORMAT [illegible] LOAD [illegible]
LOAD CODE	[illegible] the address [illegible] when [illegible]
LOAD SCREEN$	LOAD CODE 16384,6912

LPRINT	Like PRINT but using [illegible] Use the FORMAT LPRINT [illegible] about the Centronics PRINTER or RS232 [illegible] to set expansion of [illegible] into the PRINTER [illegible] and codes below [illegible] are to be [illegible] command FORMAT LPRINT "U" [illegible] ing LPRINT if printer output has been set to RS232 (using the command FORMAT LPRINT "R" [illegible] then LPRINT can be [illegible] characters to a remote [illegible]
MERGE	[illegible] LOAD [illegible] ale old program lines or [illegible] for new ones with the same [illegible] LOAD the filename may [illegible] If a drive letter is [illegible] drive [illegible] be used
MOVE [illegible] TO [illegible]	[illegible] to [illegible] Both files [illegible] and [illegible] must be on the same drive
MOVE [illegible] TO [illegible]	The string [illegible] may be "+P" "+S" "+A" "-P" "-S" "-A" [illegible] attributes of the file [illegible] + or unset (-) The attribute [illegible] protection (P) system [illegible] S [illegible] A The CAT EXP [illegible] rent settings [illegible] saved over or have any [illegible] nge them in any way performed [illegible] from the normal [illegible] by the CAT EXP [illegible] provided for compatibility [illegible] and has [illegible] their relevance to [illegible] +3
NEW	[illegible] letting any program and [illegible] up to and including the [illegible] RAMTOP [illegible] SP and PIP are [illegible] pening menu but do [illegible] AMdisk)
NEXT	[illegible] Finds the [illegible] (u) Adds [illegible] > = [illegible] > [illegible] < [illegible] < [illegible] 2 [illegible] 1 [illegible] variable in FOR [illegible]

OPEN #;	"S" "K" "P" PRINTER RS232 FORMAT LPRINT Invalid I/O device
OUT ,	B <= <= - <= <=
OVER	K
PAPER	INK
PAUSE	
PLAY , ,	B <= <= MIDI
PLOT ; ,	OVER and INVERSE PLOT B <= <= <= <=
POKE ,	B <= <= - <= <=

PRINT #n,	PRINT PRINT #n, "S" PRINT PRINT <= >= E +or - following .Ø3 or Ø.3 Ø ? AT , AT TAB PAPER INK FLASH BRIGHT INVERSE OVER
RANDOMIZE	RANDOMIZE Ø

RANDOMIZE n	[illegible] RND [illegible] <> [illegible] [illegible] <=n<=[illegible]
READ v1,v2, ... vn	[illegible] DATA [illegible] Error C [illegible] Error E [illegible] DATA [illegible]
REM	[illegible] ENTER [illegible] REM [illegible] treated [illegible]
RESTORE	RESTORE 0
RESTORE n	[illegible] DATA [illegible] to the first DATA [illegible] [illegible] DATA [illegible] DATA [illegible] READ [illegible]
RETURN	[illegible] GO SUB [illegible] [illegible] 7 [illegible] GO SUB [illegible] RETURN
RUN	RUN 0
RUN n	CLEAR [illegible] GO TO [illegible]
SAVE d	[illegible] COPY ERASE MOVE [illegible] T [illegible] SAVE [illegible]
SAVE "	[illegible] Error F [illegible]

SAVE f LINE m	Saves the program and variables so that if they are loaded, there is an automatic jump to line m
SAVE f DATA i()	Saves the numeric array i() to the file f.
SAVE f DATA i$()	Saves the character array i$() to the file f.
SAVE f CODE m,n	Saves n bytes starting at address m
SAVE f SCREEN$	SAVE f CODE 16384,6912 Saves the current screen display
SPECTRUM	Switches from +3 BASIC into 48 BASIC, maintaining any program in RAM. There is no switch back to +3 BASIC Note that ROM/RAM switching is not disabled when entering 48 BASIC using this command, (this is not the case when the option 48 BASIC is selected from the opening menu)
STOP	Stops the program with report 9 The CONTINUE command will resume the program from the following statement
VERIFY :	Like LOAD (from tape) but the tape information is not loaded into RAM - instead, it is just compared against what is already in RAM If the filename specifies a disk file (or if the current default drive is A: or B:), then no action is taken Error R if the comparison shows different bytes

Part 32
Binary and hexadecimal

Subjects covered...

Number systems
Bits and bytes

This section describes how computers count using the binary system.

Most European languages count using a more or less regular pattern — in English, for example, although it starts off a bit erratically, it soon settles down into regular groups:

twenty, twenty one, twenty two... twenty nine
thirty, thirty one, thirty two... thirty nine
forty, forty one, forty two... forty nine

... and so on, and this is made even more systematic with the numerals that we use. However, the only reason for using ten (the *decimal* system) is that we happen to have ten digits on our hands (fingers and thumbs).

Instead of using the decimal system — based on ten, computers use a form of binary called ***hexadecimal*** (or 'hex' for short) which is based on sixteen. As there are only ten digits available in our number system we need six extra digits to do the counting. So we use A, B, C, D, E and F. And what comes after F? Well, just as we, with ten fingers, write 10 for ten (a hand full), so computers use 10 for sixteen. Comparing counting in decimal to hex:

DECIMAL	HEX
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11

continued

DECIMAL	HEX
25	19
26	1A
27	1B
...	
31	1F
32	20
33	21
etc	
158	9E
159	9F
160	A0
161	A1
etc	
255	FF
256	100
and so on	

If you are using hex notation and you want to make the fact quite plain then write 'h' at the end of the number, and say 'hex'. For instance, for one hundred and fifty eight (decimal), write '9Eh' and say 'nine E hex'.

You may be wondering what all this has to do with computers. In fact computers behave as though they had only two digits, represented by a low voltage (or off) known as 0, and a high voltage (or on) known as 1. This is called the binary system, and the two binary digits are called *bits* - so a bit is either 0 or 1.

So to expand the previous table, counting to [illegible] in binary:

DECIMAL	HEX	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
...etc...		

It is customary to 'pad out' binary numbers with leading zeros so that they always contain at least four bits - for example, 0000 0000, [illegible] (representing 0 to 3 decimal).

Converting between binary and hex is very easy (use the previous table to help you).

To convert a binary number to hex, split the binary number into groups of four bits (starting at the right of the number) and convert each group of four bits into its corresponding hex digit. Finally, put the hex digits together to form the complete hex number. For example, to convert 10110100 binary into hex, convert the first (right-hand) group of four bits (0100) to 4 hex, then convert the next group of four bits (1011) to B hex, put them together, and you have the complete hex number - B4h. If the binary number is longer than eight bits, you [illegible] continue converting each group of four bits into one hex digit. For example, 1110101110000 binary [illegible] corresponds to 3AF0h.

To convert a hex number to binary, change each hex digit into four bits (again, starting at the right) then put the bits together to form the complete binary number. For example, to convert F3h to binary, first convert 3 which corresponds [illegible] 0011 binary (remember - you must use zeros to make the binary number four bits long), then convert F which corresponds to 1111 binary, put them together, and you have the complete binary number - 11110011.

Although computers use a pure binary system, humans often write the numbers stored inside a computer using hex notation - after all, the number 3AF0h (for example) is far more likely to be easily and correctly read than 0011101011110000 in sixteen bit binary notation.

The bits inside the computer are mostly grouped into sets of eight - these are called ***bytes***. A single byte can represent any number from 0 to 255 decimal (11111111 binary or FFh).

Two bytes can be grouped together to make what is technically called a ***word***. A word can be expressed using sixteen bits or four hex digits, and represents a number from [illegible] to 65535 decimal (1111111[illegible]1111 binary or FFFFh).

A byte is always eight bits, but words vary in length from computer to computer.

The `BIN` notation (used in part 14 of this chapter) provides a means of entering numbers in binary on the ***+3***. Ie. `BIN 10` represents 4 decimal, `BIN 111` represents 7 decimal, `BIN 11111111` represents 255 decimal, and so on.

You can only use 0s and 1s for this, so the number must be a non-negative whole number - for instance, you can not use `BIN -11` to represent -3 decimal, but you can use `-BIN 11` instead. The number must also be no greater than decimal 65535, ie. it can't have more than sixteen bits. If you pad out a binary number with leading zeros, for example `BIN 00000001`, the `BIN` function will mainly ignore them, and treat the number as if it were `BIN 1`.

Part 33
Example programs

Programs...

- Renumber
- Clock
- Bustout
- Telly tennis

Renumber

This short program is an aid to the renumbering facility provided by the edit menu's **Renumber** option. If you **MERGE** this program into the program you are developing (or wish to renumber), you will be able to select both the starting line number and the step size (between successive program lines).

Type **RUN 9000** to run the program, enter the start line (in the range 1...9999), enter the step size (in the range 1...9999), then press the **EDIT** key and select the **Renumber** option from the edit menu.

```
9000 INPUT "Start line",st
9010 INPUT "Step size",sp
9020 LET hst= INT (st/256)
9030 LET hsp= INT (sp/256)
9040 POKE 23413,st-256*hst
9050 POKE 23414,hst
9060 POKE 23415,sp-256*hsp
9070 POKE 23416,hsp
9080 PRINT "Press EDIT then sele
     ct Renumber option"
```

Clock

This program sets up the *+3* as an analogue (and digital) clock.

Type **RUN** to start the program, enter the hour (in the range 1...12) and enter the minute (in the range 0...59). The clock will then start.

```
10 DIM s(60): DIM c(60)
20 BORDER 0: PAPER 0: BRIGHT 1
   : INK 7: CLS
30 PRINT AT 10,1;"Hold on whil
   e I calculate"
```

```
 40 PRINT AT 11,2;"some sines a
    nd cosines"
 50 GO SUB 370
 60 LET z$="00"
 70 CLS
 80 INPUT "What hour is it ";h
 90 INPUT "How many minutes pas
    t ";m
100 LET s=0: POKE 23672,0: POKE
     23673,0
110 IF h=12 THEN LET h=0
120 LET xc=112: LET yc=90: LET
    r=70: LET rh=r/2: LET rm=r*
    3/4: LET rs=r*5/6
130 CIRCLE xc,yc,r
140 INK 1
150 FOR i=0 TO 359 STEP 30
160 PLOT (r+1)*s(i/6+1)+xc,(r+1
    )* c(i/6+1)+yc
170 NEXT i
180 INK 4
190 OVER 1: GO SUB 500
200 GO SUB 470
210 GO SUB 440
220 LET tm= INT (( PEEK 23672+2
    56* PEEK 23673)/50)
230 IF s+1=tm THEN LET os=s: LE
    T s=s+1: GO TO 250
240 GO TO 220
250 IF s=60 THEN LET s=0: POKE
    23672,0: POKE 23673,0: LET
    om=m: LET m=m+1: GO TO 290
260 PLOT xc,yc: DRAW rs*s(os+1)
    ,rs*c(os+1)
270 GO SUB 440
280 GO TO 220
290 IF m=60 THEN LET m=0: LET o
    h=h: LET h=h+1: GO TO 330
300 PLOT xc,yc: DRAW rm*s(om+1)
    ,rm*c(om+1)
310 GO SUB 470
320 GO TO 260
330 IF h=12 THEN LET h=0
```

```
340 PLOT xc,yc: DRAW rh*s(oh*5+
    1),rh*c(oh*5+1)
350 GO SUB 500
360 GO TO 300
370 PRINT AT 14,0
380 FOR i=6 TO 360 STEP 6
390 PRINT ".";
400 LET s(i/6)= SIN ((i-6)* PI
    /180)
410 LET c(i/6)= COS ((i-6)* PI
    /180)
420 NEXT i
430 RETURN
440 PLOT xc,yc: DRAW rs*s(s+1),
    rs*c(s+1)
450 LET s$= STR$ (s): PRINT OVE
    R 0; AT 18,27; INK 4;":"; I
    NK 6;z$( TO 2- LEN (s$));s$
460 RETURN
470 PLOT xc,yc: DRAW rm*s(m+1),
    rm*c(m+1)
480 LET m$= STR$ (m): PRINT OVE
    R 0; AT 18,24; INK 2;":"; I
    NK 5;z$( TO 2- LEN (m$));m$
490 RETURN
500 PLOT xc,yc: DRAW rh*s(h*5+1
    ),rh*c(h*5+1)
510 LET ph=h: IF ph=0 THEN LET
    ph=12
520 LET h$= STR$ (ph): PRINT OV
    ER 0; INK 3; AT 18,22;"  "(
     TO 2- LEN (h$));h$
530 RETURN
```

Bustout

This program provides a [illegible] and entertaining little game for one player against the computer.

To play the game type **RUN** then press any key to start.

Options:

Cursor left ( ) moves the bat left.
Cursor right ( ) moves the bat right.
The space bar trades a life for a new screen.
See if you can get the highest hiscore.

Note the following when typing in the listing

1 The "BBBBBBB "s shown in lines 30 and [illegible] are graphics characters. They are produced by pressing the **GRAPH** key once (to switch graphics mode on), typing the characters (using the [illegible] key), then pressing the **GRAPH** key again (to switch graphics mode off)

2 The "3333"s shown in line 210 are also graphics characters. Again, they are produced by pressing **GRAPH** once, pressing the **3** key four times, then pressing **GRAPH** again. (Note that these characters will look like black blocks on the screen.)

3 The "A" shown in line 430 is also a graphics character. Again, it is produced by pressing **GRAPH** once, pressing the **A** key once, then pressing **GRAPH** again

```
 10 BORDER 0: INK 0: PAPER 0: C
    LS : BRIGHT 1
 20 GO SUB 560
 30 LET b$="BBBBBBBBBBBBBBBBBB
    BBBBBBBBB": REM 28 Bs
 40 LET s$="
                    ": REM 32 spac
    es
 50 PRINT AT 3,12; INK 7; FLASH
     1;"BUSTOUT"; FLASH 0; AT 6
    ,9; INK 1;"B"; INK 7;" = 20
     Points"; AT 8,9; INK 4;"B"
    ; INK 7;" = 10 Points"; AT
    10,9; INK 2;"B"; INK 7;" =
     5 Points"
 60 PRINT AT 14,1; INK 4;"Press
     SPACE or FIRE to trade"; A
    T 16,3;"a life for a new sh
    eet."
 70 PAUSE 200
 80 LET hiscore=0
 90 LET tscore=0
100 LET lives=5
110 LET score=0
120 CLS
130 INK 7: PLOT 12,13: DRAW 0,1
    60: DRAW 230,0: DRAW 0,-160
    : INK 0
140 PRINT AT 1,2; INK 1;b$; AT
    2,2; INK 4;b$
```

```
150 FOR r=5 TO 6: PRINT AT r,2;
     INK 2;b$: NEXT r
160 LET bx=9
170 PRINT AT 19,5; INK 6;"PRESS
     ANY KEY TO START"; AT 17,4
    ;"Use < and > to move bat"
180 PAUSE 0
190 PRINT AT 19,5; INK 0;s$( TO
     24); AT 20,0;s$( TO 32); A
    T 17,4;s$( TO 24)
200 PRINT AT 21,0; INK 0;s$( TO
     32): GO SUB 540: GO TO 220
210 PRINT AT 20,bx; INK 0;" ";
    INK 5;"3333"; INK 0;" ": RE
    TURN
220 LET xa=1: LET ya=1: IF INT
    ( RND *2)=1 THEN LET xa=-xa
230 GO SUB 210
240 LET x=bx+4: LET y=11: LET x
    c=x: LET yc=y
250 REM main loop
260 IF score>1100 THEN GO TO 11
    0
270 IF INKEY$ =" " OR INKEY$ ="
    0" THEN IF lives>1 THEN LET
     lives=lives-1: GO TO 110
280 LET xc=x+xa: LET yc=y+ya
290 REM scan the keyboard
300 GO SUB 470
310 IF yc=20 THEN IF ATTR (yc,x
    c)=69 THEN PLAY "N1g": LET
    ya=-ya: LET yc=yc-2: IF xc=
    bx+1 OR xc=bx+4 THEN LET xa
    =-xa: LET xc=x+xa
320 IF yc=21 THEN PLAY "O3N7#d"
    : PRINT AT y,x;" ": GO TO 4
    50
330 GO SUB 470
340 IF yc=20 THEN GO TO 430
350 LET t= ATTR (yc,xc)
360 IF t=71 THEN GO TO 410
370 IF t=64 THEN GO TO 420
```

```
380 LET ya=-ya: LET xz=xc: LET
    yz=yc: LET yc=yc+ya: GO SUB
     510: IF t=66 THEN PLAY "N1
    e": LET score=score+5: LET
    tscore=tscore+5: GO SUB 540
    : GO TO 350
390 IF t=68 THEN PLAY "N1c": LE
    T score=score+10: LET tscor
    e=tscore+10: GO SUB 540: GO
     TO 350
400 IF t=65 THEN PLAY "N1a": LE
    T score=score+20: LET tscor
    e=tscore+20: GO SUB 540: GO
     TO 350
410 LET xa=-xa: LET xc=xc+2*xa:
     PLAY "N1f"
420 IF yc=1 THEN LET ya=1
430 PRINT AT y,x; INK 0;" "; AT
     yc,xc; INK 3;"A": LET x=xc
    : LET y=yc
440 GO TO 250
450 LET lives=lives-1: IF lives
    =0 THEN GO TO 530
460 GO SUB 540: GO TO 220
470 LET a$=INKEY$
480 IF (a$= CHR$ (8) OR a$="6")
     AND bx>1 THEN LET bx=bx-1:
     GO SUB 210: RETURN
490 IF (a$= CHR$ (9) OR a$="7")
     AND bx<25 THEN LET bx=bx+1
    : GO SUB 210: RETURN
500 RETURN
510 IF yz=20 THEN RETURN
520 PRINT AT yz,xz; INK 0;" ":
    RETURN
530 GO SUB 540: PRINT AT 10,10;
     INK 7;"GAME OVER"; AT 12,8
    ;"You scored : ";tscore: FO
    R i=1 TO 300: NEXT i: GO TO
     90
540 IF tscore>hiscore THEN LET
    hiscore=tscore
```

```
550 PRINT AT 21,11; INK 6;"HISC
    ORE ";hiscore; AT 21,1;"SCO
    RE ";tscore; AT 21,24;"LIVE
    S ";lives: RETURN
560 FOR i= USR "a" TO USR "b"+7
570 READ b
580 POKE i,b
590 NEXT i
600 RETURN
610 REM ball
620 DATA 0,60,126,126,126,126,6
    0,0
630 REM brick
640 DATA BIN 11111111
650 DATA BIN 10000001
660 DATA BIN 10111101
670 DATA BIN 10111101
680 DATA BIN 10111101
690 DATA BIN 10111101
700 DATA BIN 10000001
710 DATA BIN 11111111
```

Telly tennis

Th[illegible] +3 [illegible] computer [illegible]

Type RUN [illegible] 1 2 [illegible] to play

Options

[illegible] A [illegible] Z [illegible]
[illegible] K [illegible] M [illegible]
[illegible]

[illegible]

[illegible] "66" [illegible]
GRAPH [illegible] 6 [illegible]
[illegible] GRAPH [illegible]
[illegible]

[illegible] "8" [illegible]
[illegible] GRAPH [illegible] CAPS SHIFT [illegible] 8 [illegible]
[illegible] GRAPH [illegible]
[illegible]

3. The "A" shown in [illegible] character. Again it is produced by pressing **GRAPH** once, pressing the **A** key [illegible] pressing **GRAPH** again.

```
 10 PAPER 4: INK 0: BRIGHT 0: B
    ORDER 4
 20 CLS
 30 GO SUB 730
 40 DIM x(2): DIM y(2): DIM p(2
    )
 50 LET comp=1: LET sc1=0: LET
    sc2=0: LET z$="0"
 60 PRINT AT 2,9; INK 7;"TELLY
    TENNIS"
 70 PRINT AT 8,3;"ONE OR TWO PL
    AYERS (1/2)?"
 80 LET i$=INKEY$
 90 IF i$="1" THEN PRINT AT 12,
    8;"Use A to go up"; AT 14,8
    ;"and Z to go down": GO TO
    120
100 IF i$ ="2" THEN PRINT AT 10
    ,3;"Player 1 use A to go up
    "; AT 12,12;"and Z to go do
    wn"; AT 14,3;"Player 2 use
    K to go up"; AT 16,12;"and
    M to go down": LET comp=0:
    GO TO 120
110 GO TO 80
120 FOR i=0 TO 200: NEXT i
130 LET x(1)=2: LET y(1)=3
140 LET x(2)=29: LET y(2)=18
150 LET e$="8": LET f$="66"
160 PRINT AT 1,0;
170 GO SUB 400: REM top edge
180 FOR i=3 TO 19
190 PRINT AT i,0; INK 6;f$; INK
     0; TAB 30; INK 6;f$
200 NEXT i
210 PRINT AT 20,0;
220 GO SUB 400: REM bottom edge
230 PRINT AT 0,0; INK 1;"Player
     1: 00"; AT 0,19; INK 2;"Pl
    ayer 2 : 00"
```

```
240 LET n= INT ( RND *2)
250 FOR i=1 TO 2: PRINT AT y(i)
    ,x(i); INK i;"8"; AT y(i)+1
    ,x(i);"8": NEXT i
260 IF n=0 THEN LET xb=21: LET
    dx=1: GO TO 280
270 LET xb=19: LET dx=-1
280 LET yb=12: LET dy= INT ( RN
    D *3)-1
290 GO SUB 440: REM move bats
300 LET oxb=xb: LET oyb=yb: LET
     scd=0
310 GO SUB 580: REM move ball
320 PRINT AT oyb,oxb; INK 0;" "
330 PRINT AT yb,xb; INK 7;"A"
340 IF scd=0 THEN GO TO 290
350 PRINT AT yb,xb; INK 0;" "
360 GO SUB 380
370 GO TO 240
380 PRINT AT 0,10; INK 1;z$( TO
     2- LEN ( STR$ (sc1)));sc1;
     AT 0,30; INK 2;z$( TO 2- L
    EN ( STR$ (sc2)));sc2
390 RETURN
400 FOR i=1 TO 64
410 PRINT INK 5;e$;
420 NEXT i
430 RETURN
440 LET a$=INKEY$
450 IF a$="a" THEN LET p(1)=-1
460 IF a$="z" THEN LET p(1)=2
470 IF comp=1 THEN LET p(2)=(2*
    (y(2)<(yb))-(y(2)>(yb))): G
    O TO 50
480 IF a$="k" THEN LET p(2)=-1
490 IF a$="m" THEN LET p(2)=2
500 FOR i=1 TO 2
510 LET a= ATTR (y(i)+p(i),x(i)
    )
520 IF p(i)=2 THEN LET p(i)=1
530 IF a=32 THEN PRINT INK 0; A
    T y(i),x(i);" "; AT y(i)+1,
    x(i);" ": LET y(i)=y(i)+p(i
    )
```

```
540 PRINT AT y(i),x(i); INK i;
    "&"; AT y(i)+1,x(i);"8"
550 LET p(i)=0
560 NEXT i
570 RETURN
580 LET w= ATTR (yb+dy,xb+dx)
590 IF w=32 THEN LET xb=xb+dx:
    LET yb=yb+dy: RETURN
600 IF w=33 OR w=34 THEN LET dx
    =-dx: PLAY "V1507N1g": LET
    dy= INT ( RND *3)-1: RETURN
610 IF w=38 THEN GO TO 640
620 IF w=37 THEN PLAY "V1507N1c
    ": LET dy=-dy
630 RETURN
640 PLAY "O3V15#d": IF dx>0 THE
    N LET sc1=sc1+1: GO TO 660
650 LET sc2=sc2+1
660 LET d=(sc1=15)+2*(sc2=15):
    LET scd=1
670 IF d <> 0 THEN GO SUB 380:
    PRINT INK 7; AT 10,8;"Playe
    r ";d;" wins."; AT 12,7;"Pl
    ay again (y/n)?": GO TO 690
680 RETURN
690 IF INKEY$ ="" THEN GO TO 69
    0
700 IF INKEY$ ="y" THEN RUN
710 IF INKEY$ ="n" THEN STOP
720 GO TO 690
730 FOR i=0 TO 7
740 READ n
750 POKE USR "a"+i,n
760 NEXT i
770 RETURN
780 DATA 0,60,126,126,126,126,6
    0,0
```

Chapter 9
Using the calculator

Subjects covered...

- Selecting the calculator
- Entering numbers
- Running total
- Using built-in mathematical functions
- Editing the screen
- Assigning variables
- User defined functions
- Exit-ing from the calculator

The +3 can be used as a multifunction calculator.

To use the calculator call up the operating menu and select the **Calculator** option. (If you don't know how to select a menu option, refer back to chapter 3.)

The calculator may be selected at any time as the +3 is switched on. Alternatively if you are working on a +3 BASIC program, when you select the calculator you can select the **Exit** option from the edit menu (which returns you to the operating menu) at which point you can select the **Calculator** option. Note that any BASIC program which was running previously when you selected the calculator will be remembered and restarted when you exit from the calculator and return to +3 BASIC.

When you have selected the **Calculator** option the screen will change to...

...and the +3's calculator is ready to accept your first entry. Type it

```
6+4
```

As soon as you press **ENTER** the answer **10** will appear. (Note that you *don't* key in = as you would on a conventional calculator.)

You will see that the cursor is positioned to right of the answer, which is a ***running total*** (like on a conventional calculator). This means that you can simply type in the next operation to be carried out on the running total (without having to type in a whole new calculation). So, with the cursor still positioned to the right of the **10** on the screen, type in:

```
/5
```

..and back comes the answer 2. Now type in:

```
*PI
```

This produces the result **6.2831853** on the screen. The ***+3*** has used its built-in π function - all you had to do was type in **PI**. This applies to all the ***+3***'s mathematical functions. To demonstrate, type in:

```
*ATN 60
```

..which gives the result **9.7648943** (you may also see the [illegible] on the screen). To demonstrate, move the cursor (using the cursor [illegible] key) to the beginning of the line and then type in **INT** so that the line reads:

```
INT 9.7648943
```

..and as soon as **ENTER** is pres[illegible] gives the answer **9**. This also demonstrates that the ***+3*** doesn't *have to* perform a [illegible]lation in order to print the value of an expression. As another example, press **ENTER** then type:

```
1E6
```

..and back will come the value of that expression. Notice that before you typed in **1E6** you pressed **ENTER** on its own - this tells the ***+3*** that you are about to start a new calculation.

One extremely useful feature of the ***+3***'s calculator is that it will allow you to [illegible] to variables and then use them in subsequent calculations. This is achieved [illegible] the **LET** statement (as you would in BASIC). To demonstrate, press **ENTER** and type in the following:

```
LET x=10
```

(You must then press **ENTER** twice on the **+3** to accept the variable assignment.) Now verify that the variable x is remembered by typing:

```
x+90
```

then:

```
+x*x
```

If you are using [illegible] BASIC [illegible] the default [illegible] used by the [illegible]

BASIC [illegible]

When you [illegible] **EDIT** [illegible]

[illegible] **Exit** [illegible] **+3** BASIC [illegible] the **+3 BASIC** [illegible] **Calculator** [illegible]

[illegible] **DEF FN** [illegible] **+3** [illegible]

```
9000 DEF FN c(n)=n*n*n
```

[illegible] **FN c(n)** [illegible] **+3** [illegible] **+3** [illegible]

```
FN c(3)
```

[illegible] 27 [illegible]

Chapter 10
Peripherals for your +3

Subjects covered...

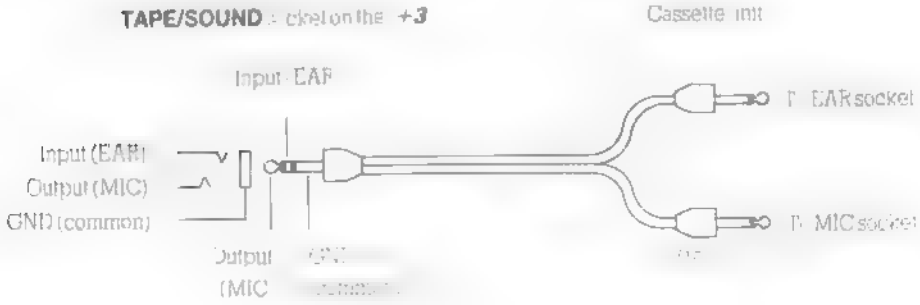
- Cassette unit
- Printer
- Additional disk drive
- Joystick(s)
- VDU Monitor
- Amplifier
- Serial devices
- MIDI device
- Auxiliary interface
- Expansion devices

The ***+3*** is [illegible] (***peripherals***) such as joystick(s) printer, cassette [illegible] essary to connect these

Cassette unit

Programs may be loade[illegible] is which instruct the computer to direct data f[illegible] tial)

To [illegible] your cassette unit to the ***+3*** you will require a suitable interconnecting lead wired as follows:



You will see that the shaft of one of the jack plugs is divided into 3 separate metal sections - this is the plug that should be inserted into the socket marked **TAPE/SOUND** at the back of the ***+3***

The shafts of the other two jack plugs are divided into only 2 separate metal sections - these are the plugs that should be inserted into the sockets on your cassette unit marked MIC and EAR. (On most commercially available leads the plug for the MIC socket is coloured red.)

(On some cassette units the MIC socket may be labelled COMPUTER IN or INPUT. Likewise the EAR socket may be labelled COMPUTER OUT or OUTPUT.)

It is important to remember that the successful transfer of programs to and from tape is largely dependent on the correct setting of the LEVEL or VOLUME control on your cassette unit. If you cannot load or save programs easily [illegible] experimenting with different LEVEL control positions until the optimum setting is found. If you cannot seem to load or save any programs at all try reversing the plugs to the MIC and EAR sockets on your cassette unit.

Details of tape operation will be found in chapter 4 and chapter 8 parts 20 and [illegible]

Printer

The ***+3*** may be used with any Centronics compatible parallel printer. We would particularly recommend the AMSTRAD DMP range of printers (eg. the DMP2000, DMP3000, DMP3160 or DMP4000) for use with the ***+3***

If you intend to [illegible] the AMSTRAD DMP2000 to the ***+3*** simply use the interconnecting lead provided with the printer.

If you wish to use any other Centronics compatible printer you will require the AMSOFT PL3 printer interconnecting lead.

Connect the end of the lead which is fitted with the flat edge connector plug into the socket marked **PRINTER** at the back of the ***+3***

Connect the other end of the lead (which is fitted with a Centronics style plug) into the socket on the printer. If your printer is [illegible] into the cut-outs at the side of the printer [illegible]

PRINTER

PIN	FUNCTION	PIN	FUNCTION
[illegible]	[illegible]	[illegible]	[illegible]

All [illegible] +3 s PRINTER [illegible]

Note that pr[illegible] use with the +3 [illegible]

The +3 [illegible]

[illegible] printer [illegible] RS232/MIDI [illegible] +3

[illegible]

Additional disk drive

The AMSTRAD model FD-1 may be added to the *+3* system as an additional disk drive.

Thanks to the versatility of *+3* BASIC you can do all necessary file maintenance, copying, erasing, etc., on a single disk drive. However, a second drive will certainly speed up these processes and reduce the scope for accidents.

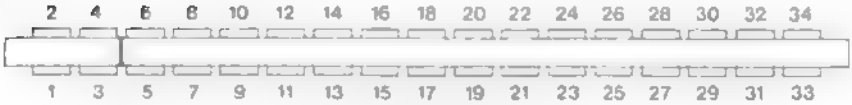
To connect the FD-1 to the *+3* you will require the AMSOFT DL 2 disk interconnecting lead.

Connect the end of the lead which is fitted with the larger edge-connector plug into the socket marked **DISK B:** at the back of the *+3*.

Connect the other end of the lead which is fitted with a smaller plug into the socket at the back of the FD-1 disk drive.

Important - Before connecting or disconnecting the additional disk drive, make sure that any disks are removed from both drives. [illegible] ctions are altered w[illegible] the system is on, it is likely that the system will crash, losing any program or data. Always save any valuable programs before meddling with connections!

When the FD-1 is connected to the *+3*, first [illegible]h on the FD-1 (using the slide switch at the back of the disk drive). Then switch on the *+3* (by plug[illegible] in the PSU). Both the green and red indicators on the front panel of the FD-1 should be illuminated. The two-drive system will then be ready to operate.



(viewed from rear)

DISK B: socket

PIN	FUNCTION	PIN	FUNCTION
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]

[illegible]

Joystick(s)

[illegible] **+3** [illegible] types of joystick [illegible] different.

[illegible] **+3** [illegible] the **JOYSTICK 1** [illegible]

[illegible] (or [illegible] option [illegible] **+3** [illegible]

[illegible] the **+3** is switched or

PIN	FUNCTION
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

JOYSTICK 1 [illegible] JOYSTICK [illegible]

VDU Monitor

[illegible] *+3* [illegible]
[illegible] *+3* [illegible]
[illegible]

[illegible] RGB/PERITEL [illegible]
+3

PIN	SIGNAL
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

RGB/PERITEL [illegible]

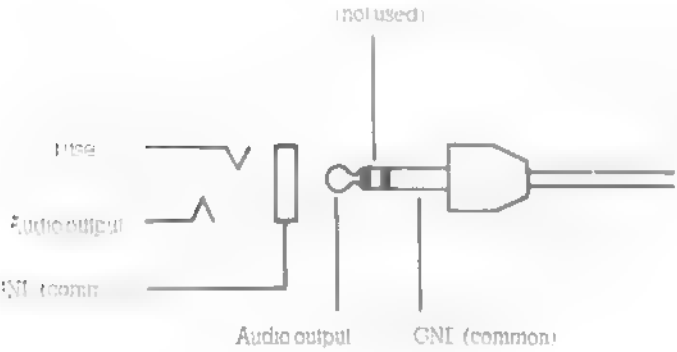
When using a monitor, some provision may have to be made for sound (if required). If the monitor has an audio input, then then this should be connected either to pin 3 of the **RGB/PERITEL** socket or to the **TAPE/SOUND** socket at the back of the *+3*. If the monitor is not capable of producing sound, then an external amplifier will have to be used. See the next paragraph for further details.

Amplifier

The *+3* normally reproduces sound through the TV set it is connected to. However, if a VDU monitor is being used, or if you would like to record or amplify the sound further, then a sound signal is available from the **TAPE/SOUND** socket at the back of the *+3*. This is a 3.5mm jack socket producing 200mV peak at approximately 3 Kohms impedance. When using an amplifier, it is worth remembering that if you have connected a cassette unit to the *+3* the tape 'load' and 'save' signals are also [illegible] the **TAPE/SOUND** socket (and therefore the amplifier's volume control should be turned [illegible] these operations).

[illegible] is to set the level of [illegible] sound produced by the **BEEP** command is set to be the same [illegible] **PLAY** [illegible] the same time. In practice, this means that **BEEP** will [illegible] that **PLAY** (which may cause problems if sound levels are critical).

[illegible] amplifier, tape recorder, etc. into the **TAPE/SOUND** socket while [illegible] *+3* [illegible]



TAPE/SOUND socket

[illegible] of the *+3* sound facilities will be found in chapter 6 part 19.

Serial devices

[illegible] *+3* [illegible] *+3* [illegible]

[illegible]

PIN	FUNCTION
[illegible]	[illegible]

RS232 [illegible]

Details of serial [illegible] operations will be [illegible] plot 3 port [illegible]

MIDI device

Although the *+3* **MIDI** [illegible] ket as the RS2[illegible] your [illegible] ead should be [illegible] nected into the [illegible] provis[illegible] the *+3* to *receive* MI[illegible] Midi [illegible] necessary before use (ex[illegible] pt the [illegible] sion of the [illegible] PLAY [illegible]

Usi[illegible] the MI[illegible]

PIN	FUNCTION
[illegible]	[illegible]

MIDI [illegible]

Details of MIDI [illegible]

Auxiliary interface

The **AUX** (auxiliary [illegible] 2 and 4). The I/O lines [illegible] the I/O lines [illegible] of the AY [illegible]

BIT	SIGNAL
[illegible]	**AUX** [illegible]
[illegible]	**AUX** [illegible]
[illegible]	**RS232** [illegible]
[illegible]	**RS232** [illegible]
[illegible]	**AUX** [illegible]
[illegible]	**AUX** [illegible]
[illegible]	**RS232** [illegible]
[illegible]	**RS232** [illegible]

Using software [illegible] out (in the same way as the **RS232/MIDI** [illegible] times [illegible] be used to drive, for example [illegible]

PIN	FUNCTION
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

AUX socket

Expansion devices

The *+3* can connect to a very wide [illegible] **EXPANSION I/O** [illegible] of the machine. Although this socket is [illegible] guarantee that a device which [illegible] *+3* [illegible] therefore before you pur[illegible] *+3* and not just with a 48K Spectrum.

> WARNING - It is very [illegible] **EXPANSION I/O** [illegible] *+3* [illegible] the *+3* and the expan[illegible]

EXPANSION I/O [illegible]

PIN	UPPER ROW (U)	LOWER ROW (L)
1	A15	[illegible]
2	A13	[illegible]
3	D7	[illegible]
4	ROM 1 $\overline{OE}$	[illegible]
5	D0	[illegible]
6	D1	[illegible]
7	D2	[illegible]
8	[illegible]	[illegible]
9	[illegible]	[illegible]
10	[illegible]	[illegible]
11	[illegible]	[illegible]
12	[illegible]	[illegible]
13	[illegible]	[illegible]
14	[illegible]	[illegible]

[illegible]

PIN	UPPER ROW (U)	LOWER ROW (L)
15	$\overline{MREQ}$	$\overline{DISKRD}$
16	$\overline{IORQ}$	$\overline{DISKWR}$
17	$\overline{RD}$	$\overline{MOTOR\ ON}$
18	$\overline{WR}$	$\overline{BUSRQ}$
19	not used	$\overline{RESET}$
20	$\overline{WAIT}$	A7
21	+12V	A6
22	-12V	A5
23	$\overline{M1}$	A4
24	$\overline{HALT}$	not used
25	A9	$\overline{BUSACK}$
26	A8	A3
27	[illegible]	A2

Details of the +3 s hardware will be found in the [illegible]

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z